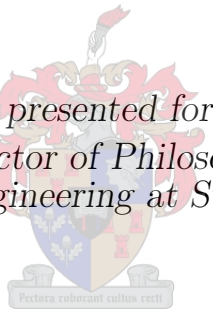
On separable primal-dual algorithms for very large-scale optimization

by

Kemal Marice Palanduz



Dissertation presented for the degree of
Doctor of Philosophy
in the Faculty of Engineering at Stellenbosch University

Promoter: Prof. Albert A. Groenwold

March 2021

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

This dissertation includes one original papers published in peer reviewed journals or books and four unpublished publications. The development and writing of the papers (published and unpublished) were the principal responsibility of myself and, for each of the cases where this is not the case, a declaration is included in the dissertation indicating the nature and extent of the contributions of co-authors

Date: 10/11/2020

Signature: Kemal M. Palanduz

Abstract

In this study, we develop two novel separable primal-dual algorithms, containing closed-form primal and dual variable expressions. The separability of the primal-dual expressions allow both algorithms to exploit massively parallel computational devices, which is desirable for very large-scale optimization. One of the algorithms is ideally suited for low-rank singular value decomposition (SVD), since the separable primal and dual updates become embarrassingly parallel for the SVD problem, allowing the algorithm to efficiently exploit general purpose graphical compute units (GPGPUs).

In the first part of this study, we develop an iterative separable augmented Lagrangian algorithm (`SALA`), which has the salient feature of embarrassingly parallel primal and dual variable expressions, hence the algorithm is ideal for implementation on massively parallel computational devices, such as GPGPUs. `SALA` solves a sequence of quadratic-like problems, able to capture reciprocal and exponential-like behavior; a desirable property in structural optimization. Since `SALA` resides in the class of alternating directions of multiplier method type algorithms, we demonstrate numerical results on structural problems requiring medium levels of accuracy.

In the second part of this study, we propose a separable Lagrangian algorithm (`SLA`) for very large-scale optimization. `SLA`, derived from the dual of Falk, solves a sequence of quadratic-like problems and, like `SALA`, is able to capture reciprocal and exponential-like behavior. `SLA` has embarrassingly parallel primal updates, while the dual variables require the solution of a positive-definite linear system. Indeed, both primal and dual variable updates can exploit massively parallel computational devices. We demonstrate numerical results for structural problems involving hundreds of millions of variables and constraints, solved for in a few minutes on a single quad-core machine.

Following the development of `SLA`, we address the low-rank SVD problem. Two separate algorithms are developed, using a variation of `SLA` that exploits the structure of the SVD problem, resulting in embarrassingly parallel primal and dual updates. Both algorithms use a GPGPU accelerated, constrained and convex sequential approximate optimization (SAO) approach to maximize the well-known Rayleigh quotient, while addressing the difficulties inherent to state-of-the-art Krylov subspace methods, such as resilience to slowly decaying singular values and constant memory requirements. The convex SAO subproblems are conditioned using a novel scaling strategy, allowing for generic solver settings to be used across a wide range of singular value distributions. We demonstrate outstanding numerical results compared to state-of-the-art Lanczos methods, in both CPU and GPGPU implementations, which significantly reduce the time-complexity required for large-scale problems.

Finally, we propose a multi-solver approach to soften the no-free-lunch (NFL) theorems for optimization on large-scale structural problems. State-of-the-art algorithms and `SLA`, each exploiting different solution strategies, compete simultaneously for a problem solution on a single multi-core system. Numerical results demonstrate the efficacy of using a multi-solver approach over a range of test problems, since said approach outperforms any standalone solver tested in terms of mean solution time.

Opsomming

In hierdie studie ontwikkel ons twee nuwe skeidbare primaal-duale algoritmes, wat analitiese primale en duale veranderlike uitdrukkings bevat. Die skeibaarheid van die primaal en duale veranderlike uitdrukkings laat albei algoritmes toe om kragtige parallelle rekenaar stelsels te benut, wat belangrik is vir grootskaalse optimering. Een van die algoritmes is by uitstek geskik vir die ontbinding van enkelwaardes van lae rangorde (SVD), aangesien die skeibare primale en duale opdaterings onafhanklik parallel word vir die SVD-probleem, wat die algoritme in staat stel om algemene grafiese rekenaareenhede (GPGPU's) doeltreffend te benut.

In die eerste deel van hierdie studie ontwikkel ons 'n iteratiewe skeibare toegevoegde Lagransiese algoritme (`SALA`), wat die opvallendste kenmerk het van onafhanklike parallelle en duale veranderlike opdaterings, daarom is die algoritme ideaal vir implementering op kragtige parallelle rekenaar stelsels, soos GPGPU's. `SALA` los 'n reeks kwadratiese probleme op, wat resiproke en eksponensiële gedrag kan vasvang; 'n wenslike eienskap in strukturele optimering. Aangesien `SALA` in die klas van alternatiewe rigtings van vermenigvuldigingsmetode (ADMM) algoritmes voorkom, toon ons numeriese resultate op strukturele probleme wat medium akkuraatheidsvlakke benodig.

In die tweede deel van hierdie studie stel ons 'n skeibare Lagransiese algoritme (`SLA`) voor vir grootskaalse optimering. `SLA`, afgelei van die duale stelling van Falk, los 'n reeks kwadratiese probleme op en is, soos `SALA`, in staat om resiproke en eksponensiële gedrag vas te vang. In `SLA` word die duale veranderlikes verkry deur die oplossing van 'n positief-definiete lineêre stelsel. Beide die primale en duale veranderlike opdaterings kan groot parallelle rekenaar stelsels gebruik. Ons demonstreer numeriese resultate vir strukturele probleme met honderde miljoene veranderlikes en beperkings wat binne 'n paar minute op 'n enkele verwerker opgelos is.

Na die ontwikkeling van `SLA` en `SALA`, spreek ons die lae rang SVD-probleem aan. Twee afsonderlike algoritmes word ontwikkel vir onderskeidelik digte en yl matrikse, met behulp van 'n variasie van `SLA` wat die struktuur van die SVD-probleem benut, wat onafhanklike parallelle primale en duale opdaterings tot gevolg het. Albei algoritmes gebruik 'n GPGPU-versnelde konvekse SAO-benadering om die bekende Rayleigh-kwosiënt te maksimeer, terwyl die probleme aangespreek word met moderne Krylov-subruimte-metodes. Die konvekse SAO-subprobleme word gekondisioneer deur gebruik te maak van 'n nuwe skaalings-strategie, wat dit moontlik maak om generiese instellings oor 'n wye verskeidenheid enkele waardeverdelings te gebruik. Ons demonstreer uitstekende numeriese resultate in vergelyking met die nuutste Lanczos-metodes, in beide CPU- en GPGPU-implementasies van ons voorgestelde SVD-

algoritmes, wat die tydkompleksiteit wat nodig is vir lae-rang SVD op grootskaalse datastelle, aansienlik verminder.

Laastens stel ons 'n multi-oplosser-benadering voor om die NFL-stellings te versag vir optimering van grootskaalse strukturele probleme. Moderne algoritmes en `SLA`, wat verskillende benaderings gebruik om die SAO-subprobleme op te los, ding gelyktydig mee om 'n probleemoplossing op 'n enkele meervoudige stelsel verwerkers. Numeriese resultate toon die doeltreffendheid van die gebruik van 'n multi-oplosser-benadering oor 'n reeks toetsprobleme aan, aangesien ons voorgestelde benadering beter as enige enkele oplosser is, alhoewel beperkte berekeningsbronne beskikbaar is.

Acknowledgments

First and foremost, I thank my family for their constant support in anything that I pursue. My postgraduate journey would not have been possible without you, and I will be forever thankful for the personal and academic growth in this period of my life.

Megan, your role in my life over the past few years cannot be overstated. I cannot thank you enough for your unconditional support and understanding.

Albert, your influence on my future could not have been more profound. My academic and personal growth is deeply attributed to your influence. Even though our meetings often dissolve into some random discussion, which of course are always interesting, your guidance has opened my eyes to an entirely new world. It has been an absolute pleasure working with you, and I sincerely hope that we continue to grow our collaboration and friendship for many years to come.

Contents

Declaration ii

Abstract iii

Opsomming v

Acknowledgments vii

Contents viii

List of Figures xii

List of Tables xiii

1 Introduction 1

1.1 Outline . . . 2

2 Sequential approximate optimization and duality 4

2.1 Sequential approximate optimization . . . 4

2.2 Duality . . . 8

3 Singular value decomposition 11

3.1 Full-rank singular value decomposition . . . 11

3.2 Low-rank singular value decomposition . . . 13

3.3 Geometric interpretation for the Rayleigh quotient . . . 15

4 A separable augmented Lagrangian algorithm 20

4.1 Abstract . . . 20

4.2 Introduction . . . 21

4.3 Some diagonal quadratic approximations . . . 24

4.4 The approximations used . . . 25
4.4.1 A spherical diagonal quadratic approximation (SPH-QDR) . . . 25
4.4.2 The quadratic approximation to the reciprocal approximation (T2:R) 26
4.5 Alternating directions type methods (ADMM) . . . 26
4.6 Closed-form expressions for QP-like problems . . . 27
4.7 Objective and constraint function scaling . . . 29
4.8 Subproblem scaling . . . 29
4.9 Numerical experiments . . . 30
4.10 Conclusions and recommendations . . . 32
4.11 Tables for numerical results . . . 34
4.12 Further work . . . 35
4.12.1 Exploiting constraint Jacobian sparsity . . . 35
4.12.2 GPGPU implementation . . . 35

5 A separable primal-dual algorithm **39**
5.1 Abstract . . . 39
5.2 Introduction . . . 40
5.3 Quadratic-like approximations . . . 42
5.4 A quadratic-like dual subproblem . . . 44
5.5 A quadratic-like separable Lagrangian algorithm . . . 46
5.5.1 Primal projection free subproblems . . . 46
5.5.2 Enforcing subproblem primal projections . . . 48
5.5.3 Global convergence . . . 51
5.6 A primal-dual active-set strategy . . . 52
5.7 Numerical experiments . . . 53
5.7.1 General and structural test problems . . . 54
5.7.2 Large-scale structural test problems . . . 55
5.7.3 Topology optimization . . . 55
5.8 Conclusions and recommendations . . . 57
5.9 Tables for numerical results . . . 59
5.10 GPGPU implementation . . . 62

6 A deflation based low-rank singular value decomposition algorithm **64**
6.1 Abstract . . . 64
6.2 Introduction . . . 65
6.3 Problem formulation . . . 68

6.4 Quadratic-like approximations . . . 70
6.5 Embarrassingly parallel primal-dual updates . . . 71
6.6 An implicitly restarted scaling strategy . . . 74
6.7 Numerical experiments . . . 75
6.7.1 Example 1: multiplicity and slowly decaying singular values . . . 77
6.7.2 Example 2: variably decaying singular values . . . 78
6.7.3 Example 3: PCA of real-world datasets . . . 79
6.8 Conclusions and recommendations . . . 80
6.9 Tables for numerical results . . . 81

7 A low-rank singular value decomposition algorithm 85
7.1 Abstract . . . 85
7.2 Introduction . . . 86
7.3 Problem formulation . . . 88
7.4 Quadratic-like approximations . . . 90
7.5 A closed-form SAO method . . . 92
7.6 Implicit restarting for objective function scaling . . . 95
7.7 A sparse pseudo-deflation power method . . . 97
7.8 Numerical experiments . . . 97
7.8.1 Example 1: variably decaying singular values . . . 100
7.8.2 Example 2: multiplicity and slowly decaying singular values . . . 101
7.8.3 Example 3: uniformly distributed sparse random matrices . . . 102
7.8.4 Real-world datasets . . . 103
7.9 Conclusion and recommendations . . . 105
7.10 Tables for numerical results . . . 106

8 Softening the no-free-lunch theorems for structural optimization 110
8.1 Abstract . . . 110
8.2 Introduction . . . 110
8.3 Sequential approximate optimization . . . 113
8.3.1 Separable quadratic-like approximations . . . 114
8.3.2 Diagonal QP subproblems . . . 115
8.3.3 Pure dual subproblems . . . 115
8.4 Global convergence . . . 116
8.5 Numerical experiments . . . 117
8.6 Conclusions and recommendations . . . 118

8.7 Tables for numerical results 120

9 Conclusion and recommendations **123**

9.1 Conclusions 123

9.2 Recommendations for future work 124

List of References **127**

List of Figures

2.1 The convergence path of SAO for a nonlinear problem $\mathcal{P}_{\text{NLP}}$. 5
2.2 The convergence path of SAO for a constrained nonlinear problem $\mathcal{P}_{\text{NLP}}$. . . . 8

3.1 Differing singular value distributions used for selected test problems. 15
3.2 The Rayleigh quotient manifold constrained to the unit sphere. 16
3.3 Rayleigh quotient manifolds formed by well-spaced singular values. 18
3.4 Rayleigh quotient manifolds formed by closely-spaced singular values. 18
3.5 Rayleigh quotient manifolds formed by multiplicate singular values. 19

4.1 The OpenMP and GPGPU implementations of `SALA` for Vanderplaats' #2. . . 37
4.2 The OpenMP and GPGPU implementations of `SALA` for Vanderplaats' #3. . . 37

5.1 The MBB beam topology optimization problem, using the T2:R approximation. 56
5.2 The MBB beam topology optimization problem, using the T2:R and SPH-QDR approximations. 57
5.3 A comparison of the GPGPU and CPU implementations of `SLA`. 63

6.1 The decomposition times for the multiplicate and slowly-decaying singular values. 77
6.2 The decomposition times for the variably decaying singular values. 78

7.1 The decomposition times required for the variably decaying singular values. . . 101
7.2 The decomposition times required for the multiplicate and slowly decaying singular values. 102
7.3 The decomposition times required for the random matrices. 103
7.4 A CPU and GPGPU comparison for the real-world dataset decompositions. . . 104

List of Tables

4.1 The SALA test problems. . . . 31
4.2 Numerical results for the test problems using the ALGENCAN and SALA solvers. . 34
4.3 Numerical results for the Vanderplaats' test problems using the dense and sparse SALA solvers. . . . 38
4.4 Comparison of the serial, OpenMP and GPGPU SALA implementations for the Vanderplaats' test problems. . . . 38

5.1 The general and structural test problems. . . . 54
5.2 The MBB beam topology optimization problem, using the T2:R approximation. 57
5.3 The MBB beam topology optimization problem, using the T2:R and SPH-QDR approximations. . . . 57
5.4 Numerical results for the general and structural test problems. . . . 59
5.5 The large-scale structural test problems, together with the solver solution times. 60
5.6 Numerical results for the large-scale structural test problems. . . . 61
5.7 Comparison of the GPGPU and CPU solver implementations of SLA for the Vanderplaats' test problems. . . . 63

6.1 The selected test problems. . . . 77
6.2 Numerical results for the multiplicate and slowly-decaying singular values. . . 81
6.3 GPGPU numerical results for the multiplicate and slowly-decaying singular values. 81
6.4 Numerical results for the variably decaying singular values. . . . 81
6.5 GPGPU numerical results for the variably decaying singular values. . . . 82
6.6 Numerical results for PCA. . . . 83
6.7 GPGPU numerical results for PCA. . . . 84

7.1 The selected test problems. . . . 100
7.2 Numerical results for the variably decaying singular values. . . . 106
7.3 GPGPU numerical results for the variably decaying singular values. . . . 106
7.4 Numerical results for the multiplicate and slowly decaying singular values. . . 107

7.5 Numerical results for random sparse matrices. 107
7.6 GPGPU numerical results for random sparse matrices. 107
7.7 Numerical results for large-scale real-world datasets. 108
7.8 GPGPU numerical results for large-scale real-world datasets. 109

8.1 The large-scale structural test problems. 119
8.2 Numerical results for the large-scale structural test problems, using a multi-solver strategy. 120
8.3 Standalone `SLA` numerical results. 120
8.4 Standalone `LSQP` numerical results. 121
8.5 Standalone `CPLEX` numerical results. 121
8.6 Standalone Falk dual numerical results. 122

Chapter 1

Introduction

The minimization or maximization of a scalar *objective* function, which may be a nonlinear or linear map of $f(\mathbf{x}) : \mathcal{R}^n \rightarrow \mathcal{R}$, is one of the most important tools used in the design and analysis of engineering applications. The mapping, depending upon n *primal* or *design* variables, may be subject to either equality or inequality constraints, relying on mappings $g_j(\mathbf{x}) : \mathcal{R}^n \rightarrow \mathcal{R}$, where j is the index of a given constraint. Constraints may take one of three general forms, namely nonlinear, linear or bound (or box) constraints. If no constraints are present, the problem is said to be *unconstrained*, in contrast to the *constrained* term used to denote problems subject to constraints of any form. The distinction between the two fundamental problem types is important, since a host of methods have been developed to solve either specific one, even though focus for solving constrained problems has generally been on solving some 'unconstrained' variation thereof. Herein, we mainly concern ourselves with the efficient solutions for constrained problems.

As far as this dissertation is concerned, we are interested in 'black-box' simulations that yield both function (zero-order) and first-order information, since the optimization methods we rely on require accurate first-order information to generate optimal search directions. Given n design (primal) variables, together with the aforementioned objective and constraint function mappings; the optimization aim is to modify the design variables such that they minimize the objective function, while respecting the limits defined by the constraint functions.

Instead of directly attempting to find the optimal solution for a problem, notwithstanding that it may be impossible to do so, it is generally more efficient to iteratively improve the design for large-scale problems. Appropriate criteria is then used to determine whether our iterative refinement has yielded an acceptable design, in which case we terminate the iterative process. The iterative process we strictly focus on is sequential approximate optimization (SAO). SAO has been firmly established as an optimization approach for large-scale simulation or optimization based problems, often containing computationally expensive, nonlinear objective and constraint functions [1, 2]. Furthermore, exact second-order Hessian information of the objective and constraint functions, which can be highly beneficial in generating optimal search directions, may be prohibitively expensive. A symmetric $n \times n$ Hessian matrix is required for n design variables, hence for large-scale optimization where n is in the hundreds of millions, Hessian requirements of $\mathcal{O}(n^2)$ become expensive in both memory and

computational resources.

Following from the above, simulation based optimization contains two broad computationally expensive operations. The first is performing a sensitivity analysis, which yields function and first-order information for a given set of n design variables. Computational fluid dynamics (CFD) and finite element methods (FEM) are but two examples of 'black-box' simulations that require computationally expensive sensitivity analyses. The second is solving the resulting SAO subproblem formed from a given sensitivity analysis. SAO attempts to solve these difficult, 'black-box' simulation problems by efficiently minimizing the number of sensitivity analyses performed, while requiring minimal subproblem solution effort to update the design variables. Neither of these tasks are trivial, especially for large-scale optimization involving hundreds of millions of design variables and/or constraints.

This dissertation is mainly concerned with developing efficient and scalable algorithms to solve the resulting SAO subproblems. The scalability of an algorithm is dependent on both the computational complexity of the operations required, as well as the ability to distribute said operations to parallel workers. As a simple example, consider two solvers requiring $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$ operations. Clearly, as n grows to some large number, the $\mathcal{O}(n^2)$ solver will require substantially less computational effort than the $\mathcal{O}(n^3)$ solver. Next, assume that n 'workers' are available for the $\mathcal{O}(n^2)$ solver. Ignoring overheads, if each worker evenly takes on $\mathcal{O}(n)$ work concurrently, the apparent solution time appears to be $\mathcal{O}(n)$, even though $\mathcal{O}(n^2)$ work is performed. Many solvers cannot scale by using 'parallel workers'; a consequence of certain serial algorithmic requirements. With the advent of multi-core processors, general purpose graphical compute units (GPGPUs) and distributed systems, it is imperative that solvers can scale for large problems.

After developing two solvers in this dissertation, both of which are scalable, we focus on applying one of these solvers to the singular value decomposition problem (SVD). SVD is a cornerstone in linear algebra, with wide ranging applications, some of which are described in what is to come. Essentially, the problem reduces to finding a few leading eigenvectors of a positive (semi) definite matrix, otherwise known as Rayleigh quotient maximization.

Much like the simulation based problems that SAO is effective at solving, the Rayleigh quotient maximization problem requires expensive sensitivity analyses, albeit that the analyses can be efficiently performed on parallel computational devices. Additionally, the problem suffers from prohibitively expensive second-order information, since the Lagrangian Hessian is of $\mathcal{O}(n^2)$ for a $p \times n$ matrix $\mathbf{A}$. For matrices containing millions of rows and columns, memory requirements of $\mathcal{O}(n^2)$ are often prohibitively expensive. Hence, we only make use of first-order sensitivity and approximate $\mathcal{O}(n)$ Hessian information throughout, thus substantially reducing both computational and memory requirements.

1.1 Outline

The structure of this dissertation is largely based on chapters of self-contained papers, which have either been published, are under peer review, or have been submitted for publication. At the beginning of these chapters, a short description of the paper's publication status,

along with co-authors, is given. The self-contained nature of the papers may result in some repetition across chapters, notwithstanding that we aim to keep repetition to a minimum.

We provide an introduction to SAO and the concept of duality in Chapter 2. The self-contained papers introduce SAO and duality only briefly to avoid excessive repetition, hence we provide a more in-depth explanation of both topics. Indeed, all the work herein relies heavily on SAO and duality.

In Chapter 3, we introduce the SVD of a matrix. We expand on the concepts of full-rank and low-rank SVD, before introducing the Rayleigh quotient, which we maximize to solve the low-rank SVD problem. Lastly, we present a geometric interpretation for the Rayleigh quotient, allowing for the visualization of the manifolds that our optimization methods operate on. Visualizing these manifolds helps to conceptually understand the difficulties associated with maximizing the Rayleigh quotient.

In Chapter 4, we develop a separable augmented Lagrangian algorithm (`SALA`) for optimal structural design. The separability of `SALA` is ideal for exploitation by parallel computational devices, hence the algorithm should scale well, provided sufficient parallel computing resources are available. Modern GPGPUs are ideal for exploiting the separability of `SALA`, as we demonstrate at the end of the chapter.

Chapter 5 presents our development of a separable Lagrangian algorithm (`SLA`) for very large-scale optimization. `SLA` is an extremely efficient algorithm, able to solve problems involving hundreds of millions of variables and constraints in a few minutes on a modest quad-core system. `SLA`, like `SALA`, is well-suited for parallel computing, which we again demonstrate at the end of the chapter.

Chapters 6 and 7 present two solvers, namely `saosvd-d` and `saosvd`, for efficiently computing the low-rank SVD problem. Both solvers exploit a variant of `SLA`, hence the solvers can be efficiently implemented on GPGPUs, which we extensively demonstrate in both chapters.

In Chapter 8, we propose a multi-solver, multi-core simultaneous solver execution strategy to soften the effects of the no-free-lunch (NFL) theorems for optimization. Our numerical results demonstrate the efficacy of using competing algorithms to solve a wide range of large-scale structural optimization problems, as opposed to using a standalone, single solver approach.

Finally, we present concluding remarks and recommendations for future work in Chapter 9.

Chapter 2

Sequential approximate optimization and duality

In this chapter, two topics are introduced that we extensively use throughout this dissertation, namely sequential approximate optimization (SAO) and duality. Together, these two concepts forge the development of the solvers in Chapters 4 and 5. The solver in Chapter 5 is further modified, again using the principles of SAO and duality, in Chapters 6 and 7. We introduce SAO in Section 2.1, before concluding with duality in Section 2.2

2.1 Sequential approximate optimization

SAO will be our preferred optimization method of choice throughout this study, since SAO was intended for large-scale simulation based problems, which often require computationally expensive function and first-order evaluations. Generally, expensive function evaluations result in expensive first-order information, with exact second-order information often prohibitively expensive both in evaluation and storage. For a problem with n design variables, the exact second-order storage requirements alone are of $\mathcal{O}(n^2)$. However, second-order information is extremely useful and can greatly accelerate convergence. One such example is Newton's method, which has far better quadratic local convergence than a method relying solely on first-order information, such as the linear local convergence of gradient descent.

Throughout this dissertation, local convergence will be defined as convergence by a method to a stationary point occurring from some 'region' around said stationary point. It is common for methods not to converge to said stationary point when starting outside this 'region'. In contrast, global convergence is defined as convergence by a method to a stationary point irrespective of the initial point selected, provided the initial point is feasible. In constrained optimization, we quantify a stationary point with the Karush-Kuhn-Tucker (KKT) conditions, detailed in what is to come. Within our SAO framework SAOi [3], mechanisms exist to enforce global convergence, even though a chosen solver from within the framework may not be globally convergent. These mechanisms are discussed in relevant chapters to come.

Notwithstanding that exact second-order information suffers from the so-called curse of di-

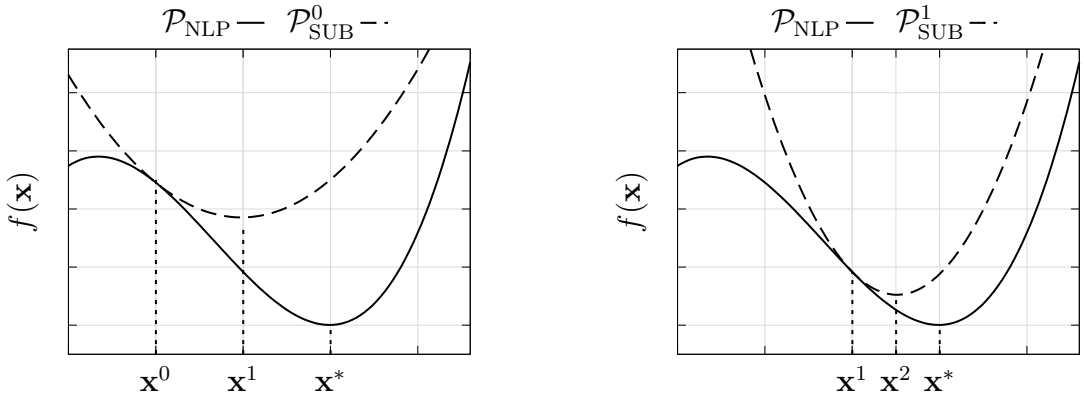
mensionality, it seems wasteful to take *no* advantage of *any* second-order information, even if the second-order information needs to be *approximated*. A popular example is the class of so-called quasi-Newton methods[1], which exploit computationally cheap approximate second-order information. Indeed, SAO attempts to address approximate second-order information in a computationally efficient manner, albeit differently to quasi-Newton methods for reasons we will discuss in what is to come.

Consider the minimization of the nonlinear optimization problem $\mathcal{P}_{\text{NLP}}$ in Figure 2.1, which depends upon the n primal (design) variables, such that

$$\min_{\mathbf{x}} \; f(\mathbf{x}), \qquad \mathbf{x} \in \mathcal{X} \in \mathcal{R}^n, \tag{2.1}$$

where we denote the primal bounds by the compact set $\mathcal{X}$. Directly solving for the solution of (2.1) is often challenging and computationally expensive, especially if n in (2.1) is large. Furthermore, a 'black-box' simulation, such as those found in computational fluid dynamics (CFD) or finite element analysis (FEM), may be used to generate (2.1), potentially resulting in expensive function and/or first-order evaluations. As previously mentioned, if first-order information is expensive, exact second-order information is generally prohibitively expensive in evaluation and storage.

Given a nonlinear optimization problem $\mathcal{P}_{\text{NLP}}$ and point $\mathbf{x}^k$ in SAO, a *convex* subproblem $\mathcal{P}^k_{\text{SUB}}$ is constructed to approximate local behavior of $\mathcal{P}_{\text{NLP}}$ at the point $\mathbf{x}^k$, as depicted in Figure 2.1. A convex function, such as $\mathcal{P}^k_{\text{SUB}}$, may be defined as satisfying the inequality



(a) $\mathcal{P}^0_{\text{SUB}}$ constructed around $\mathbf{x}^0$.

(b) $\mathcal{P}^1_{\text{SUB}}$ constructed around $\mathbf{x}^1$.

Figure 2.1: The convergence path of SAO for a nonlinear problem $\mathcal{P}_{\text{NLP}}$.

$$f(\alpha\mathbf{x} + \beta\mathbf{y}) \leq \alpha f(\mathbf{x}) + \beta f(\mathbf{y}) \quad \forall \quad \mathbf{x}, \mathbf{y} \in \mathcal{X} \in \mathcal{R}^n, \tag{2.2}$$

with $\alpha + \beta = 1$ and $\alpha, \beta \geq 0$ real valued scalars[2] [5]. The result in (2.2) is important,

[1] An efficient and popular quasi-Newton solver, which we use in chapters to come, is the limited memory Broyden-Fletcher-Goldfarb-Shanno algorithm (`l-BFGS-b`) [4].

[2] A notable special case of convexity is the class of linear programming (LP).

since it guarantees that $\mathcal{P}^k_{\text{SUB}}$ has a unique local minimum[3] for all $\mathbf{x}^k \in \mathcal{X}$, assuming that $\mathcal{X}$ is closed and bound. The SAO sequence is then formed by constructing and minimizing successive $\mathcal{P}^k_{\text{SUB}}$, which hopefully converge to some global minimum[4] $\mathbf{x}^*$ of $\mathcal{P}_{\text{NLP}}$, as shown in Figures 2.1a and 2.1b. The convergence of SAO to a global minima of $\mathcal{P}_{\text{NLP}}$, provided a mechanism is used to enforce global convergence, is well-established, with one such example provided in [7].

Although Figure 2.1 represents how SAO problems are constructed for $\mathcal{P}_{\text{NLP}}$ subject to simple bound (box) primal constraints, many problems of interest are subject to linear and/or nonlinear constraints. Hence, we approximate the local behavior of *both* the objective and constraint functions to construct $\mathcal{P}^k_{\text{SUB}}$. Perhaps the most notable feature of SAO is that the $\mathcal{P}^k_{\text{SUB}}$ are constructed from *separable* and *convex* objective and constraint functions, such that

$$\tilde{f}_j(\mathbf{x}) = \sum_{i=1}^{n} \tilde{f}_j(x_i), \quad j = 0, \ldots, m. \tag{2.3}$$

From hereon, we denote f_0 as the objective function and f_j as the $j = 1, \ldots, m$ constraint functions, with g_j alternatively used to denote the constraint functions in some parts of the text.

Popular examples of algorithms that exploit separable approximations of the form in (2.3) include the method of moving asymptotes (MMA) of Svanberg [8, 9], and the convex linearization algorithm (CONLIN) of Fleury and Braibant [10]. However, we make use of the separable approximations proposed by Groenwold and Etman [11] throughout this study, which rely on 'approximated-approximations'. Here, diagonal quadratic approximations are constructed to a number of useful other approximations, including the reciprocal approximation that relates stress to area, as well as to the previously mentioned CONLIN and MMA approximations. The variant of Groenwold and Etman [11] uses an incomplete series expansion (ISE) [2] to construct the approximate subproblems, by means of a second-order Taylor series around a point $\mathbf{x}^k$, such that

$$\tilde{f}^k_j(\mathbf{x}) = f_j(\mathbf{x}^k) + \sum_{i=1}^{n} \left(\frac{\partial f_j(\mathbf{x}^k)}{\partial x_i} \right) (x_i - x_i^k) + \frac{1}{2} \sum_{i=1}^{n} c^k_{2i_j} (x_i - x_i^k)^2. \tag{2.4}$$

Clearly, (2.4) is both separable and convex, provided the c_{2i_j} terms are strictly positive, thereby forming a *diagonal* positive-definite Hessian matrix. Hence, SAO does not require exact second-order information, and instead exploits *approximate* Hessian information by relying upon either direct or so-called intervening variables c_{2i_j}; the latter able to capture both reciprocal and exponential behavior often found in structural optimization. The exact construction of the intervening variables will be discussed in sections to come, but suffice to say they ensure (2.4) is both separable and convex, while requiring minimal computational effort to update. The intervening variables are constructed as a linear first-order Taylor series around a given approximate subproblem, and thus contain cheap analytical updates. In the

[3] Infeasible subproblems can arise and create difficulties in the SAO process, but popular methods such as relaxation exist to ameliorate such difficulties. An alternative method for dealing with infeasible subproblems may be found in [6].

[4] Assuming that $\mathcal{P}_{\text{NLP}}$ is convex. If $\mathcal{P}_{\text{NLP}}$ is non-convex, we hope for convergence to some local minima.

special case of linear programming, no direct or intervening variables are used, hence (2.4) becomes a first-order Taylor series.

Throughout this dissertation, two main approximations $c^k_{2i_j}$ will be used; namely the quadratic approximation to the reciprocal approximation (T2:R) and the spherical quadratic approximation (SPH-QDR). In general, the T2:R approximation is used for problems of a structural nature, since it captures the inverse relationship between stress and area well. Alternatively, the SPH-QDR approximation is well-suited for more general problem types, notwithstanding that the additional conservatism offered by T2:R has proved fruitful for general problems. While we mainly rely on the above mentioned approximations, detailed in further chapters, it is important to note that many different approximations exist.

To simplify notation for $\mathcal{P}^k_{\text{SUB}}$ around some k-th iteration point, let

$$f^k_j = f_j(\mathbf{x}^k),$$
$$\left(\frac{\partial f_j}{\partial x_i}\right)^k = \frac{\partial f_j(\mathbf{x}^k)}{\partial x_i},$$

so that (2.4) may be rewritten in abbreviated notation as

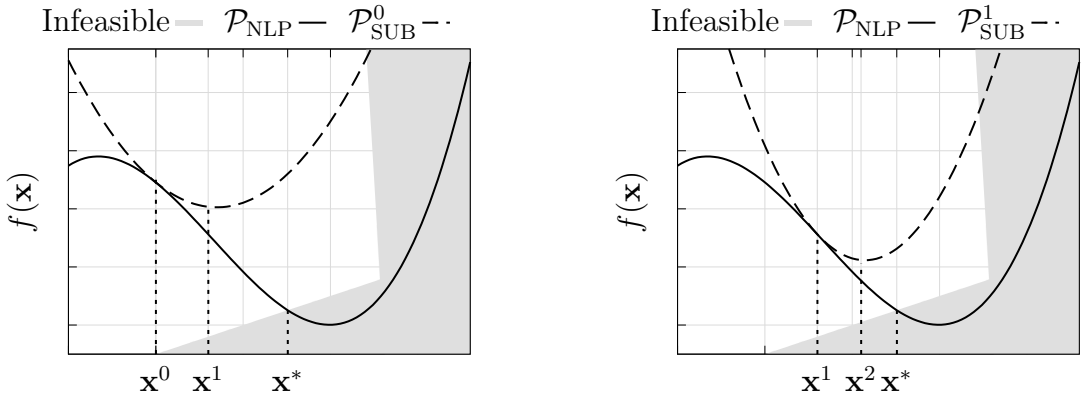
$$\tilde{f}^k_j(\mathbf{x}) = f^k_j + \sum_{i=1}^{n} \left(\frac{\partial f_j}{\partial x_i}\right)^k (x_i - x^k_i) + \frac{1}{2}\sum_{i=1}^{n} c^k_{2i_j}(x_i - x^k_i)^2, \tag{2.5}$$

which we use throughout this study. The *constrained* subproblems, at an iteration k, are then formally expressed as

$$\begin{aligned} \min_{\mathbf{x}} \ & \tilde{f}^k_0(\mathbf{x}) \\ \text{subject to} \ & \tilde{f}^k_j(\mathbf{x}) \leq 0, && j = m_e + 1, \ldots, m, \\ & \tilde{f}^k_j(\mathbf{x}) = 0, && j = 1, \ldots, m_e, \\ & \check{x}^k_i \leq x_i \leq \hat{x}^k_i, && i = 1, \ldots, n, \end{aligned} \tag{2.6}$$

where $\check{x}^k_i$ and $\hat{x}^k_i$ are the lower and upper bounds on a primal variable x_i respectively, ensuring that $\mathbf{x} \in \mathcal{X}$. Solving the constrained problem in (2.6) is generally more challenging than solving the bound constrained variant in (2.1). Numerous SAO techniques have emerged to solve (2.6), some of which will be discussed and developed in chapters to come.

To illustrate the effect of constraints on the SAO procedure, consider the case when $\mathcal{P}_{\text{NLP}}$ is subject to constraints, thereby creating an infeasible region highlighted in gray, shown in Figure 2.2. As in Figure 2.1, we construct the approximations $\mathcal{P}^k_{\text{SUB}}$, except we now make use of (2.6) for both the objective and constraint functions, and solve for the solution of the *constrained* subproblem. Despite both Figures 2.1 and 2.2 using the same $\mathcal{P}_{\text{NLP}}$, constraining $\mathcal{P}_{\text{NLP}}$ results in an *increased* objective function value, since the *feasible region* has been reduced. Therefore, the SAO process may not always produce monotonically decreasing objective functions values, since subproblem solutions that lie in the infeasible region will require that constraints be satisfied, at the expense of increased objective function values. However, the SAO iteration path is often highly dependent on the type of solver used, some of which are discussed in what is to come.



(a) $\mathcal{P}^0_{\text{SUB}}$ constructed around $\mathbf{x}^0$.

(b) $\mathcal{P}^1_{\text{SUB}}$ constructed around $\mathbf{x}^1$.

Figure 2.2: The convergence path of SAO for a constrained nonlinear problem $\mathcal{P}_{\text{NLP}}$.

2.2 Duality

The popular Lagrangian statement is extensively used throughout this dissertation to solve constrained problems. In essence, it transforms a constrained problem into an unconstrained variant, by penalizing the objective function with constraint violations. Consider a general problem given by

$$\begin{aligned} \min_{\mathbf{x}} \ & f(\mathbf{x}) \\ \text{subject to} \ & f_j(\mathbf{x}) \leq 0, \qquad j = m_e + 1, \ldots, m, \\ & f_j(\mathbf{x}) = 0, \qquad j = 1, 2, \ldots, m_e, \end{aligned} \tag{2.7}$$

where $f_j(\mathbf{x})$ denotes $j = m_e + 1, 2, \ldots, m$ *inequality* constraints and $f_j(\mathbf{x})$ denotes $j = 1, 2, \ldots, m_e$ *equality* constraints. Using the popular and well-known Lagrangian formulation, (2.7) may be expressed as

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \sum_{j=1+m_e}^{m} \lambda_j f_j(\mathbf{x}) + \sum_{j=1}^{m_e} \lambda_j f_j(\mathbf{x}), \tag{2.8}$$

where $\boldsymbol{\lambda}$ are the so-called *Lagrange multipliers* for the inequality and equality constraint functions. In certain chapters, $\boldsymbol{\mu}$ will interchangeably be used with $\boldsymbol{\lambda}$, with the intended meaning clear from context. Throughout this study, we will refer to the Lagrange multipliers as the *dual variables* instead; a common term in SAO and structural optimization literature.

Many of the most effective algorithms rely on solving (2.8), with prominent examples including sequential quadratic programming (SQP) methods, of which interior-point and active-set methods are popular examples, as well as many efficient pure dual methods, such as the aforementioned MMA and CONLIN algorithms. The primal and dual solutions to the Lagrange function in (2.8) can be found by determining the respective stationary points of (2.8). To ensure that a stationary point is unique, the objective and constraint functions must be convex. Furthermore, the Jacobian matrix, which contains the derivatives of the constraint

functions with respect to the primal variables, must be of full-rank for all binding and active constraints[5]. Then, the *sufficient* conditions for optimality may be given by the well-known KKT conditions [12], as

$$\begin{aligned}
\frac{\partial \mathcal{L}(\mathbf{x},\boldsymbol{\lambda})}{\partial x_i} &= \frac{\partial f}{\partial x_i}(\mathbf{x}^*) + \sum_{j=1}^{m} \lambda_j \frac{\partial f_j}{\partial x_i}(\mathbf{x}^*) = 0, \quad i = 1,\ldots,n, \\
\frac{\partial \mathcal{L}(\mathbf{x},\boldsymbol{\lambda})}{\partial \lambda_j} &= f_j(\mathbf{x}^*) = 0, \quad j = 1,\ldots,m_e, \\
\frac{\partial \mathcal{L}(\mathbf{x},\boldsymbol{\lambda})}{\partial \lambda_j} &= f_j(\mathbf{x}^*) \leq 0, \quad j = m_e+1,\ldots,m, \\
\lambda_j^* f_j(\mathbf{x}^*) &= 0, \quad j = m_e+1,\ldots,m, \\
\lambda_j^* &\geq 0, \quad j = m_e+1,\ldots,m.
\end{aligned} \tag{2.9}$$

If an inequality constraint is negative, i.e. $f_j(\mathbf{x}^*) < 0,\ j = m_e+1,\ldots,m$, then the associated multiplier must be $\lambda_j^* = 0$, and the constraint is said to be inactive. Conversely, if $f_j(\mathbf{x}^*) \geq 0$, then the respective multiplier must satisfy $\lambda_j^* \geq 0$ and the constraint is said to be active and binding at $\mathbf{x}^*$. In other words, an active inequality constraint is satisfied as an equality constraint at the solution point, albeit with a strictly non-negative Lagrange multiplier. For all equality constraints $f_j(\mathbf{x}^*),\ j = 1,\ldots,m_e$, the multipliers λ_j^* are always active and binding.

The Lagrangian formulation may be applied to SAO subproblems, by constructing an approximate Lagrangian from the approximate objective and constraint functions, such that at a point $\mathbf{x}^k$

$$\tilde{\mathcal{L}}^k(\mathbf{x},\boldsymbol{\lambda}) = \tilde{f}_0^k + \sum_{j=1+m_e}^{m} \lambda_j \tilde{f}_j^k(\mathbf{x}) + \sum_{j=1}^{m_e} \lambda_j \tilde{f}_j^k(\mathbf{x}). \tag{2.10}$$

Iteratively minimizing (2.10) with respect to the primal variables, followed by maximizing (2.10) with respect to the dual variables, yields the primal and dual solutions $\mathbf{x}^{k*} = \mathbf{x}^{k+1}$ and $\boldsymbol{\lambda}^{k*} = \boldsymbol{\lambda}^{k+1}$ respectively. Since $\tilde{\mathcal{L}}^k$ is constructed from convex functions, the solution point $\tilde{\mathcal{L}}^k(\mathbf{x}^{k*},\boldsymbol{\lambda}^{k*})$ is guaranteed to be unique. After solving for $\tilde{\mathcal{L}}^k(\mathbf{x}^{k*},\boldsymbol{\lambda}^{k*})$, $\tilde{\mathcal{L}}^{k+1}$ is constructed and minimized in the SAO fashion, until some predetermined convergence criteria is met, such as the aforementioned KKT conditions in (2.9). Throughout our numerical testing, we measure the Euclidean norm of the KKT conditions with $\mathcal{K}$, by letting

$$\mathcal{K} = \sqrt{\sum_{i=1}^{n} \left(\frac{\partial \mathcal{L}(\mathbf{x},\boldsymbol{\lambda})}{\partial x_i} \right)^2}. \tag{2.11}$$

In Chapters 4 and 5, we develop algorithms to minimize (2.10), or a slight variation thereof, such as the augmented Lagrangian statement presented by Rockafellar [13]. The augmented Lagrangian is expressed as

$$\tilde{L}^k(\mathbf{x},\boldsymbol{\lambda},\rho) = \tilde{f}_0^k(\mathbf{x}) + \sum_{j=1}^{m} \lambda_j \tilde{f}_j^k(\mathbf{x}) + \frac{\rho}{2} \sum_{j=1}^{m} \left(\tilde{f}_j^k(\mathbf{x}) \right)^2, \tag{2.12}$$

[5]This is the so-called *constraint qualification*, which needs to be met at the solution point to ensure the existence of the Lagrange multipliers.

where ρ is the so-called penalty parameter, *augmenting* the Lagrangian in (2.10) with any constraint violation. To enforce convexity of (2.12), it is prescribed that $\rho \geq 0$, while setting $\rho = 0$ results in identical problem statements between (2.10) and (2.12). By selecting an optimal value of ρ, it is possible that (2.12) converges faster than (2.10) for a given problem; however, many difficulties are associated with selecting an optimal ρ, which we further elaborate on in Chapter 4.

In what is to come, we develop algorithms that solve the SAO subproblems formed by (2.10) and (2.12). The details of the algorithms, together with numerical results for challenging structural problems, are discussed in the respective chapters.

Chapter 3

Singular value decomposition

Although we are interested in computing low-rank decompositions herein, we present a brief overview of the full-rank decomposition in Section 3.1, purely for the sake of clarity. In Section 3.2 we discuss the low-rank approximate decomposition, building on the foundation formed from the full-rank decomposition. Finally, we present a geometric interpretation for the Rayleigh quotient maximization problem in Section 3.3.

3.1 Full-rank singular value decomposition

Matrix decompositions, along with their practical applications, are one of the most fruitful developments in matrix theory [14]. Singular value decomposition (SVD) was discovered independently by Eugenio Beltrami and Camille Jordan more than a hundred years ago, but only transformed the field of linear algebra in the 1960's. This was predominantly due to the development of practical methods for computing it, a notable example being the Golub and Reinsch algorithm [14, 15].

Being perhaps one of the most useful decompositions, SVD has several important applications over a broad range of domains. These include the structural and sensitivity reanalyzes of modified structures [16], efficient solutions of algebraic expressions that arise in mechanical systems dynamics [17], linear algebra and linear algebra-based signal processing [18, 19, 20, 21], etc.

SVD is a mathematical problem that is considered extremely difficult and 'hard' [22]. The problem suffers from the so-called curse of dimensionality, where the computational effort grows exponentially with the size of the problem. The full-rank decomposition is computationally expensive for a large matrix $\mathbf{A} \in \mathcal{R}^{p \times n}$, with the best known algorithm requiring an $\mathcal{O}(4p^2n + 22n^3)$ floating-point operations (flops) [23].

SVD can be represented as an optimization problem with a computationally demanding sensitivity analysis, as we will show in what is to follow. Much like the simulation based problems that SAO is so effective at solving, the SVD problem suffers from prohibitively expensive second-order information; since the Lagrangian Hessian grows with $\mathcal{O}(n^2)$ for a $p \times n$ matrix $\mathbf{A}$. Hence, we only make use of first-order sensitivity throughout, using cheap

diagonal second-order information to approximate the exact Hessian.

Assuming that a real $p \times n,\ n \geq p$ matrix $\mathbf{A}$ exists

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p1} & a_{p1} & \dots & a_{pn} \end{bmatrix},$$

then an $n \times n$ square covariance matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A} \in \mathcal{R}^{n \times n}$ can be formed that is symmetric and positive-semidefinite; whereby all eigenvalues are real and larger than or equal to zero [24]. For the complex case when $\mathbf{A} \in \mathcal{C}^{p \times n}$, the Hermitian transpose is used instead, such that $\mathbf{A}^{\mathrm{H}}\mathbf{A} \in \mathcal{R}^{n \times n}$ forms the real covariance matrix. Since the complex case is a trivial extension of the real case, only the real case will be considered throughout this study.

Let $\lambda_1, \lambda_2, \dots, \lambda_n$ be the eigenvalues of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$, then the *singular values* of $\mathbf{A}$ may be defined as [24]

$$s_1 = \sqrt{\lambda_1}, \quad s_2 = \sqrt{\lambda_2}, \quad \dots, \quad s_n = \sqrt{\lambda_n}. \tag{3.1}$$

Assuming that our matrix has at least more columns than rows, i.e. $n \geq p$, $\mathbf{A}$ can be written in the form of an exact SVD, given by

$$\mathbf{A} = \mathbf{W}\mathbf{S}\mathbf{V}^{\mathrm{T}}. \tag{3.2}$$

$\mathbf{W}$ is an $p \times p$ orthonormal matrix, whose columns are the left-singular vectors of $\mathbf{A}$, $\mathbf{S}$ is a $p \times n$ rectangular diagonal matrix with the descending singular values (3.1), and $\mathbf{V}$ is an $n \times n$ orthonormal matrix whose columns are the right-singular vectors of $\mathbf{A}$ [23, 25]. Represented as matrices, they appear as

$$\mathbf{S} = \begin{bmatrix} s_1 & & & & & \\ & s_2 & & & & \\ & & \ddots & & & \\ & & & s_p & 0 & \dots & 0 \end{bmatrix}, \quad \mathbf{W} = \begin{bmatrix} | & & | \\ \mathbf{w}_1 & \dots & \mathbf{w}_p \\ | & & | \end{bmatrix}, \quad \mathbf{V} = \begin{bmatrix} | & & | \\ \mathbf{v}_1 & \dots & \mathbf{v}_n \\ | & & | \end{bmatrix}.$$

For $n < p$, the SVD of $\mathbf{A}$ exists with $\mathbf{S}$ slightly modified, such that

$$\mathbf{S} = \begin{bmatrix} s_1 & & & \\ & s_2 & & \\ & & \ddots & \\ & & & s_n \\ & & & 0 \\ & & & \vdots \\ & & & 0 \end{bmatrix},$$

and $s_{n+1} = \dots = s_p = 0$ [24].

The right-singular vectors, $\mathbf{V}$, are the eigenvalues of the covariance matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$; an important property that we exploit throughout our optimization approach. This can be proven

from the well-known eigenvector and eigenvalue relationship [26], together with the definition of the decomposition in (3.2), where

$$\begin{aligned}\left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)\mathbf{V} &= \left(\mathbf{W}\mathbf{S}\mathbf{V}^{\mathrm{T}}\right)^{\mathrm{T}}\left(\mathbf{W}\mathbf{S}\mathbf{V}^{\mathrm{T}}\right)\mathbf{V} \\ &= \left(\mathbf{V}\mathbf{S}^{\mathrm{T}}\mathbf{W}^{\mathrm{T}}\right)\left(\mathbf{W}\mathbf{S}\mathbf{V}^{\mathrm{T}}\right)\mathbf{V} \\ &= \mathbf{V}\mathbf{S}^{\mathrm{T}}\left(\mathbf{W}^{\mathrm{T}}\mathbf{W}\right)\mathbf{S}\left(\mathbf{V}^{\mathrm{T}}\mathbf{V}\right) \\ &= \mathbf{V}\mathbf{S}^{\mathrm{T}}\left(\mathbf{I}\right)\mathbf{S}\left(\mathbf{I}\right) \\ &= \mathbf{V}\left(\mathbf{S}^{\mathrm{T}}\mathbf{S}\right). \end{aligned} \tag{3.3}$$

The relation in (3.3) proves that the singular values are indeed the square root of the eigenvalues of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$, as per the definition in (3.1). Proceeding from the definition of SVD in (3.2), the *rank* of a matrix can be defined as the number of its nonzero singular values. Thus, it follows that an exact decomposition of the original matrix still holds if the singular values that are zero are discarded, along with their respective singular-vectors. Similarly, the *closest approximation* to the original matrix, for a given rank, can be constructed by using a chosen number of the largest singular values. Since the number of singular values used defines the rank of the approximate matrix, the so-called *low-rank approximation* is used to denote an approximate matrix containing far fewer singular values than the rank of the original matrix.

3.2 Low-rank singular value decomposition

Throughout this study, we shall define the rank of a matrix as σ, which will be particularly useful for our rank-σ matrix approximations. We now redefine the matrices $\mathbf{W} \in \mathcal{R}^{p\times\sigma}, \mathbf{S} \in \mathcal{R}^{\sigma\times\sigma}$ and $\mathbf{V} \in \mathcal{R}^{n\times\sigma}$ in Section 3.1 as each being of rank-σ, allowing us to use consistent notation throughout. The *singular-triplets* of a rank-σ matrix are then the combination of $\{\mathbf{w}_r, s_r, \mathbf{v}_r\}$, $r = 1, \ldots, \sigma$, with r denoting the index of a particular singular-triplet. Furthermore, a *mode* will be used to interchangeably describe a singular-triplet or a singular value, with the context clear on which we refer to.

For low-rank approximations, it is common to construct an approximate matrix $\mathbf{A}_\sigma$ of rank σ, given by

$$\mathbf{A}_\sigma = \sum_{r=1}^{\sigma} \mathbf{w}_r s_r \mathbf{v}_r^{\mathrm{T}} = \mathbf{W}\mathbf{S}\mathbf{V}^{\mathrm{T}}.$$

The closest approximation $\mathbf{A}_\sigma$ can be found by means of the Eckart-Young-Mirsky theorem [25]

$$\min_{\mathbf{A}_\sigma} \|\mathbf{A} - \mathbf{A}_\sigma\|_F,$$

where $\|\cdot\|_F$ denotes the Frobenius norm. In general, we consider cases where $\sigma \ll \mathrm{rank}(\mathbf{A})$, and solve for the low-rank approximate matrix as an optimization problem. The optimization problem may be reformulated as the maximization of the Rayleigh quotient [27, 28], which will be our optimization problem of choice throughout this study.

Low-rank approximations have significantly lower computational complexity compared to the $\mathcal{O}(4p^2n + 22n^3)$ flops [23] required for the complete decomposition; with examples found

in state-of-the-art Krylov methods [29, 30], which require $\mathcal{O}(\sigma^3 pn)$ flops. This makes low-rank approximations particularly attractive for many practical large-scale datasets, where a significant fraction of the total variance in $\mathbf{A}$ can be explained by a few of the leading modes (in general). Although a number of methods exist to compute an approximation $\mathbf{A}_\sigma$, theoretical proof ensures that for a desired decomposition accuracy, SVD will find the optimal $\mathbf{A}_\sigma$ [23]. This renders SVD as the *de facto* choice for any method computing $\mathbf{A}_\sigma$.

Unsurprisingly, the multiplicity and spacing between successive singular values has a significant impact on the computational effort required to solve for $\mathbf{A}_\sigma$. Multiplicity, or the number of times a singular value is repeated, and slowly decaying singular values result in slow convergence rates and are thus difficult to solve for; even with state-of-the-art low-rank approximation methods [29, 30]. A relatively simple analogy explaining the difficultly of the problem can be shown from the well-known power-method [31], since the SVD problem is an eigenvalue and eigenvector problem of the covariance matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$. The power-method, along with state-of-the-art low-rank Lanczos and Jacobi methods, exploits the so-called *Krylov* subspace, defined as [31]

$$K^z(\mathbf{B}, \mathbf{b}_1) := \mathrm{span}(\mathbf{b}_1, \mathbf{B}\mathbf{b}_1, \ldots, \mathbf{B}^{z-1}\mathbf{b}_1).$$

Assuming that $\mathbf{B} \in \mathcal{R}^{n\times n}$ is a symmetric matrix and $\mathbf{b}_1 \in \mathcal{R}^n$ is uniformly randomly distributed $\in (-1, 1)$, the power-method will converge to the largest singular value with high probability; assuming that a large enough value of z is chosen [31]. However, in the case of *any* method exploiting Krylov information, the *rate* of convergence is dependent on the ratio and multiplicity of the successive singular values being solved for. Multiplicity and closely spaced singular values adversely impact convergence, with detailed proofs found in [31]. In essence, the distribution of the singular values cannot be overstated from an algorithmic performance perspective. Indeed, the size of the matrix does affect performance; but from a theoretical perspective the rate of convergence is unaffected for Krylov methods [31].

Together with the presented information and the excellent research in [29], we carefully select distributions of singular values for our numerical testing. The distributions are presented in Figure 3.1, and will be used consistently throughout this study. The (normalized) Marchenko-Pastur distribution in Figure 3.1 results from the construction of random matrices, which forms the basis for random matrix theory and has many important applications, most of which can be found in [32]. The first mode is generally easy to solve for, since it is well-separated from the second largest mode; with the following modes significantly more challenging to solve for as the remaining $s_2, \ldots, s_\sigma$ decay slowly. For the variably decaying singular values [29], a term we coin because the difference between the singular values sharply decreases after a predetermined mode, we test a solvers ability to solve for well-spaced singular values as well as values that decay slowly. The decomposition of these matrices becomes significantly more challenging after $r = 20$, again shown in Figure 3.1.

Perhaps the most challenging test, especially for the Lanczos methods, is the distribution that contains multiplicity followed by slowly decaying singular values [29]. For methods exploiting Krylov subspaces, multiplicity represents a unique challenge, since the eigenvectors found for a particular singular value may be a linear combination of the eigenvectors unique to each of the multiplicate singular values. Many techniques have been proposed to dampen these issues, such as restarting and preconditioning, but they are an inherent problem to

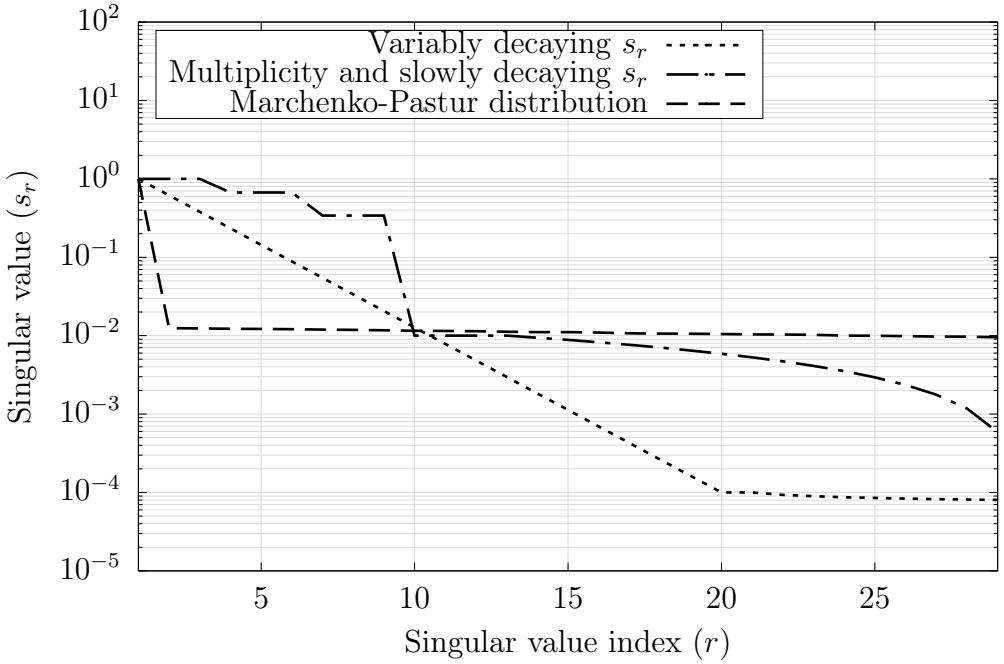


Figure 3.1: Differing singular value distributions used for selected test problems.

Krylov subspace methods [29, 30]. Since methods exploiting Krylov subspaces have become state-of-the-art in low-rank decompositions, we use this particular distribution to test our proposed method(s) to a known challenging problem.

3.3 Geometric interpretation for the Rayleigh quotient

Inspired by the research in [28], we provide a brief geometric interpretation for the Rayleigh quotient maximization problem [27, 28], since it is our SVD optimization problem of choice throughout this study. We discuss the problem in detail in sections to come, but for this section it will suffice to understand that the SVD problem is a maximization of some constrained or unconstrained objective function. Consider the constrained nonlinear optimization problem, given by

$$\begin{aligned} &\max_{\mathbf{x}} \; f(\mathbf{x}) = \mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x} \\ &\text{subject to } \mathbf{x}^{\mathrm{T}}\mathbf{x} = 1, \end{aligned} \tag{3.4}$$

where $\mathbf{x}^*$ is the desired leading right-singular vector of $\mathbf{A}$, or the eigenvector of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$. The feasible region formed by (3.4) for some matrix $\mathbf{A} \in \mathcal{R}^{2\times 2}$ is depicted as a 'lasso' in Figure 3.2. Clearly from (3.4), the search space is confined to the unit sphere, hence a unit circle is formed from the projection of (3.4) onto the plane $x_1 - x_2$, depicted in Figure 3.2. Although we discuss the convexity of (3.4) in what is to come, both maximum points of (3.4) in Figure 3.2 are identical in magnitude and represent valid eigenvectors, corresponding to the desired singular value of $\mathbf{A}$. In Chapter 6, Figure 3.2 and (3.4) represents our low-rank SVD

problem of choice, while Chapter 7 uses the aforementioned representation for solving the leading singular-triplet.

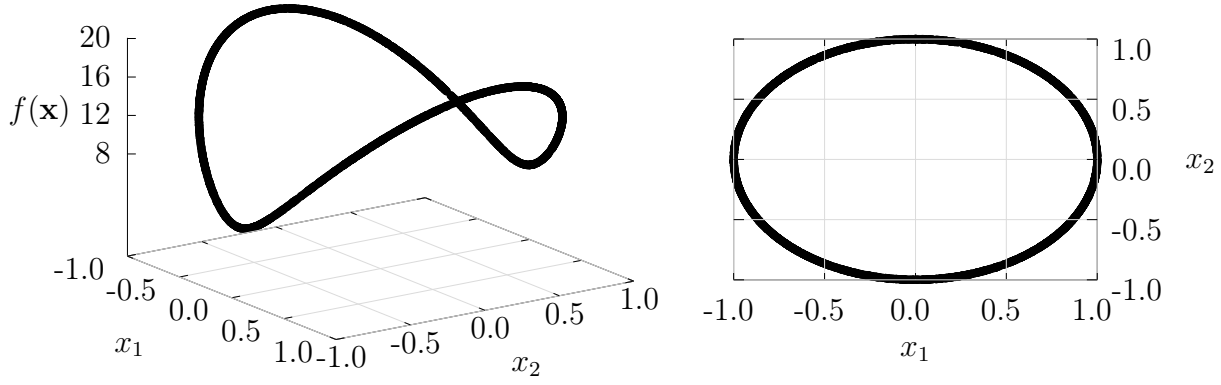


Figure 3.2: The Rayleigh quotient manifold constrained to the unit sphere (left), together with the corresponding unit circle formed from the projection onto the $x_1 - x_2$ plane (right).

By explicitly normalizing the primal variables in the objective function, (3.4) is equivalent to the unconstrained Rayleigh quotient problem [27, 28], given by

$$\max_{\mathbf{x}} f(\mathbf{x}) = \frac{\mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x}}{\|\mathbf{x}\|_2^2}, \tag{3.5}$$

which again yields the leading mode. Although (3.5) is undefined when $\|\mathbf{x}\|_2 = 0$, this never occurs in practice, provided a reasonable initial point is selected for the optimization algorithm. Hence, we make no further mention of the case when $\|\mathbf{x}\|_2 = 0$. To solve for further modes, we employ deflation or linear orthogonality constraints, discussed in chapters to come.

Suffice it to say, for a $p \times n$ matrix $\mathbf{A}$, (3.4) is a mapping $f(\mathbf{x}) : \mathcal{R}^n \rightarrow \mathcal{R}$, and under the given constraint, forms the constraint surface known as the Stiefel manifold [28], shown in Figure 3.2. As previously mentioned, the surface of this manifold is constrained to the unit sphere, from the explicit constraint that $\mathbf{x}^{\mathrm{T}}\mathbf{x} = 1$. However, since we implicitly incorporate the constraint that the vector must lie on the unit sphere in (3.5), by normalizing the primal variables in the formulation itself, a manifold (surface) that is not constrained to the unit sphere $\in \mathcal{R}^n$ may be constructed.

Although we never maximize an unconstrained variant of the Rayleigh quotient, the manifolds formed by (3.5) are helpful in visualizing the optimization challenges of finding stationary points on said manifolds. For $\mathbf{A} \in \mathcal{R}^{2\times 2}$, the manifold formed by the leading mode in (3.5), denoted by M_1, can be visualized. For the smallest mode, we use deflation in (6.9) and plot the manifold formed by the deflated matrix, denoted by M_2.

The maximum points on either manifold, i.e. $\max_{\mathbf{x}} f(\mathbf{x})$, is the square of the respective singular value for that manifold. Additionally, the minimum point on M_1 coincides with the

maximum point on M_2, which is the direction of least variance in $\mathbf{A}^{\mathrm{T}}\mathbf{A}$. Indeed, this holds true for higher-dimensional spaces, where (3.5) may be minimized to obtain the eigenvector associated with the least variance[1]. Furthermore, the manifolds of all modes except the leading mode have a minimum point of zero, representing a direction in which no variance exists, since the variance has been removed through either orthogonality constraints or deflation.

It is important to discuss the convexity of the manifolds formed in Figures 3.2, 3.3, 3.4 and 3.5. Firstly, we begin by noting the symmetry found in the aforementioned figures, since

$$\begin{aligned} f(\mathbf{x}) &= f(-\mathbf{x}) \\ \frac{\mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x}}{\|\mathbf{x}\|_2^2} &= \frac{(-\mathbf{x}^{\mathrm{T}})\mathbf{A}^{\mathrm{T}}\mathbf{A}(-\mathbf{x})}{\|-\mathbf{x}\|_2^2}. \end{aligned} \tag{3.6}$$

Secondly, the stationary points of (3.4) and (3.5), $\mathbf{x}^*$ or $-\mathbf{x}^*$, correspond to desired eigenvectors [27]. The twin peaks in the maximization problem are simply explained by a given eigenvector differing only in sign; a result of the aforementioned symmetry. Hence, throughout this study, it suffices to treat problems (3.4) and (3.5) as convex, which significantly simplifies our optimization procedures. For more detail on the convexity of problems (3.4) and (3.5), which is equivalent to determining the Euclidean or spectral norm of $\mathbf{A}$, we refer the interested reader to References [5, 27, 33].

Figure 3.3 represents the manifolds formed by a matrix $\mathbf{A} \in \mathcal{R}^{2\times 2}$ containing a 'large' and 'small' singular value, representing the problem of well-spaced singular values. M_1 has well-defined contours, with clear maximum stationary points, and is generally easy to solve for with most methods. In contrast, M_2 has a relatively 'flat' surface, with low projected gradients across the entire manifold. Due to the low projected gradients, our numerical testing indicates that optimization algorithms often falsely converge on such manifolds, or take many iterations to find the maximum point; a notorious problem in optimization [34]. Figure 3.4 shows the manifolds formed by a matrix with two closely-spaced singular values of 'medium' magnitude, representing problems subject to slowly decaying singular values. Since the singular values are of reasonable magnitude, the contours in both manifolds are well-defined. However, the stationary points are less defined than those of well-spaced singular values, requiring more iterations for convergence. If the singular values were to be of 'small' magnitude, the problem becomes much more challenging, since the manifold becomes significantly 'flatter'. Multiplicity is represented in Figure 3.5. The level manifold formed by M_1 is somewhat misleading for higher-dimensional spaces, since any arbitrary choice of $\mathbf{x}$ will not necessarily be a maximum point (i.e. it is not a 'level' manifold in high-dimensional space). However, the two-dimensional case required a diagonal matrix with identical values to produce multiplicate singular values, thus creating a level manifold. Nevertheless, visualizing the manifolds is still useful. The solution vector, $\mathbf{x}^*$, corresponding to the maximum singular value may be any of the solution vectors for the multiplicate sites. Therefore, although the shape of M_2 is unique, the orientation is not and is dictated by the choice of solution vector on M_1.

[1] In the case of $\mathbf{A} \in \mathcal{R}^{2\times 2}$, the smallest mode is equal to the second largest mode, which is of course not the case in higher-dimensional spaces; hence in higher-dimensional spaces the minimum point on M_1 is not associated with the second largest mode.

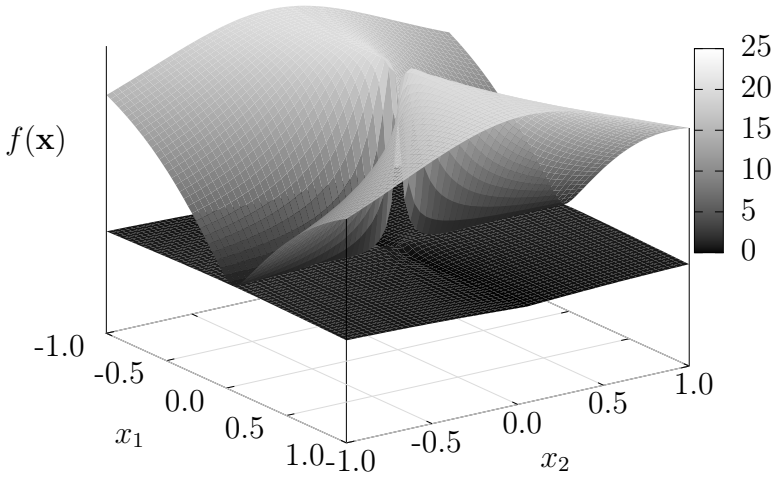


Figure 3.3: Rayleigh quotient manifolds formed by well-spaced singular values.

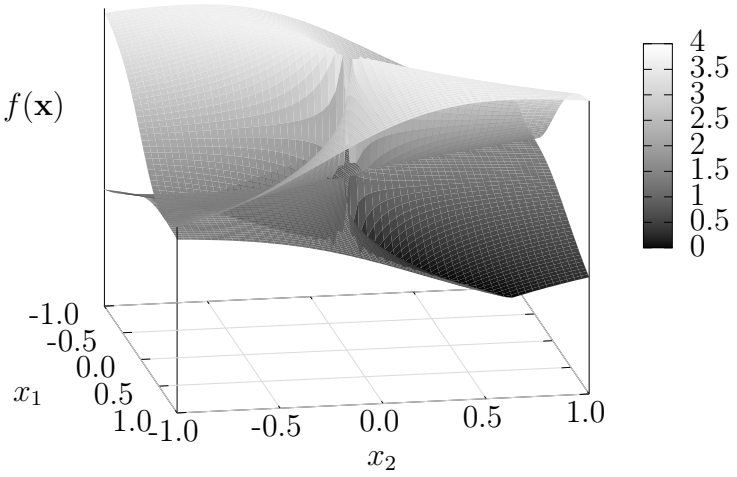


Figure 3.4: Rayleigh quotient manifolds formed by closely-spaced singular values.

Before departing from this section, we shall briefly discuss the notion of subspace iteration methods, which all state-of-the-art Lanczos methods tested herein exploit. Subspace iteration methods solve for all σ desired singular-triplets 'simultaneously', in contrast to the approach we propose herein, which requires that each singular-triplet is solved for individ-

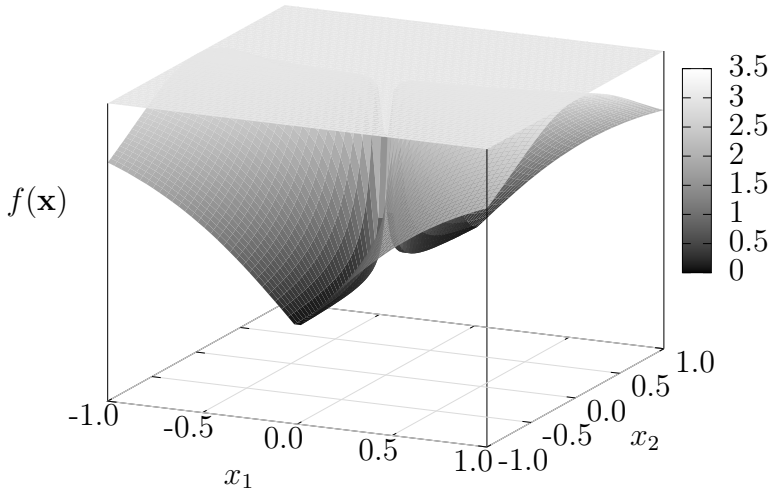


Figure 3.5: Rayleigh quotient manifolds formed by multiplicate singular values.

ually. There are many benefits to a subspace approach, especially from a computational standpoint, since low-rank SVD methods often require many matrix-vector operations on $\mathbf{A}^{\mathrm{T}}\mathbf{A}$. However, our proposed approach is necessary to exploit special structures in the low-rank SVD problem, which in turn results in highly efficient algorithms, as we demonstrate in chapters to come.

Chapter 4

A separable augmented Lagrangian algorithm

The work presented here, excluding Section 4.12, originates from a paper titled "A separable augmented Lagrangian algorithm for optimal structural design" [35]. The paper is co-authored by Prof. Albert A. Groenwold.

4.1 Abstract

We propose an iterative separable augmented Lagrangian algorithm (SALA) for optimal structural design, with SALA being a subset of the alternating directions of multiplier method (ADMM) type algorithms. Our algorithm solves a sequence of separable quadratic-like programs, able to capture reciprocal and exponential-like behavior, which is desirable in structural optimization. A salient feature of the algorithm is that the primal and dual variable updates are all updated using closed-form expressions.

Since algorithms in the ADMM class are known to be very sensitive to scaling, we propose a scaling method inspired by the well-known ALGENCAN algorithm. Comparative results for SALA, ALGENCAN and the Galahad LSQP solver are presented for selected test problems.

Finally, although we do not exploit this feature herein, the primal and dual updates are both embarrassingly parallel, which makes the algorithm suitable for implementation on massively parallel computational devices, including general purpose graphical processor units (GPGPUs).

4.2 Introduction

In this study, we consider an equality constrained nonlinear optimization problem $\mathcal{P}_{\text{NLP}}$ of the form

$$\begin{aligned} \min_{\mathbf{x}} \ & f_0(\mathbf{x}) \\ \text{subject to} \ & f_j(\mathbf{x}) = 0, \qquad j = 1, 2, \ldots, m, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, \qquad i = 1, 2, \ldots, n, \end{aligned} \tag{4.1}$$

where $f_0(\mathbf{x})$ is a real valued scalar objective function and the $f_j(\mathbf{x})$, $j = 1, 2, \cdots, m$ are m equality constraint functions, which depend on the n real (design) variables $\mathbf{x} = \{x_1, x_2, \cdots, x_n\}^T \in \mathcal{X} \subset \mathcal{R}^n$, with l_i and u_i respectively the lower and upper bounds on variable x_i.

The functions $f_0(\mathbf{x})$ and $f_j(\mathbf{x})$ are assumed to be (at least) once continuously differentiable.

Although many, or possibly even most interesting problems in structural optimization are normally formulated using inequality constraints only[1], it is convenient to herein use only equality constraint functions, for reasons that will become clear in sections to follow. Accordingly, we will assume that any inequality constraints present may be reformulated as equality constraints, with the aid of so-called slack variables s_j.

In this approach, an inequality constraint $f_j(\mathbf{x}) \leq 0$ is rewritten as

$$f_j(\mathbf{x}) + s_j = 0,$$

subject to $s_j \geq 0$. Similarly, a separable inequality constraint can be rewritten as

$$f_{ji}(x_i) + s_{ji} = 0,$$

subject to $s_{ji} \geq 0$. This simple technique is not only well-known, but also often used, even in successful commercial codes, e.g. see [36, 37]. For the sake of brevity, we will in the following not even mention the presence of the slack variables s_j or s_{ji} for inequality constraints; their use is implied.

Arguably, the state-of-the-art in structural optimization is to use a sequential approximate optimization (SAO) algorithm to solve problem $\mathcal{P}_{\text{NLP}}$. SAO relies on the iterative solution of a sequence of approximate optimization problems $\mathcal{P}_P[k]$, $k = 0, 1, 2, \cdots$. In turn, the approximate optimization problems, or subproblems, are normally based on approximation functions $\tilde{f}_j^k(\mathbf{x})$ which have a relatively *simple structure*, albeit that they may all be nonlinear. Then, at some iterate $\mathbf{x}^k$, we obtain continuous primal approximate subproblem $\mathcal{P}_P[k]$, written as

$$\begin{aligned} \min_{\mathbf{x}} \ & \tilde{f}_0^k(\mathbf{x}) \\ \text{subject to} \ & \tilde{f}_j^k(\mathbf{x}) = 0, \qquad j = 1, 2, \ldots, m, \\ & \check{x}_i^k \leq x_i \leq \hat{x}_i^k, \qquad i = 1, 2, \ldots, n. \end{aligned} \tag{4.2}$$

[1] A notable exception being the so-called simulated analysis and design (SAND) methodology.

This primal problem contains n unknowns, m equality constraints, and $2n$ side or bound constraints (not counting any slack or relaxation variables that may have been introduced). The side constraints are normally handled in an efficient way which does not require additional computational effort.

In SAO, a most notable feature of primal approximate subproblem $\mathcal{P}_P[k]$ is that the approximation functions $\tilde{f}^k(\mathbf{x})$ and $\tilde{f}_j^k(\mathbf{x})$ are *separable*. Possibly surprising at first, this is routinely done since the evaluation and storage of second order information in structural optimization is considered prohibitively expensive on the computational devices available to us today. Instead, so-called intermediate or intervening variables are relied upon which, when substituted into a linear Taylor series expansion, reveal behavior that is representative of the underlying physics of nonlinear optimization problem $\mathcal{P}_{\text{NLP}}$ (assuming that the physics is understood in the first place). The approximations then become linear in the intervening variables used.

In structural optimization for example, the reciprocal intermediate variables $z_i = x_i^{-1}$ are important and often used, since they capture the inverse relationship between stress and area well. Invariably, the intermediate variables used themselves are separable; when substituted into a linear or first-order Taylor series expansion, this of course in turn results in separable approximations and hence, separable approximate subproblems $\mathcal{P}_P[k]$.

Examples of popular algorithms that use separable approximations include the convex linearization algorithm (CONLIN) of Fleury and Braibant [10], and its generalization, the method of moving asymptotes (MMA) of Svanberg [8, 9]. Groenwold and Etman [11] have proposed the use of separable diagonal quadratic approximations, which rely on the so-called 'approximated-approximations' approach to capture reciprocal-like behavior [38]. In this approach, the quadratic approximation to the reciprocal approximation itself is constructed, or even the quadratic approximation to the CONLIN and MMA approximations already mentioned.

Given the dominance of separable approximations, it is convenient to rewrite subproblems (4.2) as

$$\begin{aligned} \min_{\mathbf{x}} \quad & \sum_{i=1}^{n} \tilde{f}_0(x_i) \\ \text{subject to} \quad & \sum_{i=1}^{n} \tilde{f}_{ji}^k(x_i) = 0, \qquad j = 1, 2, \ldots, m, \\ & \check{x}_i^k \leq x_i \leq \hat{x}_i^k, \qquad i = 1, 2, \ldots, n. \end{aligned} \tag{4.3}$$

It is prudent to here mention that the SAO subproblems used in structural optimization are often solved in the dual space, which is problem-free from a theoretical point of view, if the approximations are *convex* and separable (although solution of the subproblems may still be demanding). The use of pure dual methods is particularly popular when inequality constraints only are present.

Of course, primal separability does not imply that the associated *dual problem* often favored in structural optimization will be separable; in general, this is indeed not the case. Hence, in pure dual methods, we have the disappointing situation that even though the primal

approximations are all separable, the maximization over the dual variables is not separable. (Notwithstanding, the primal-dual relationships may of course benefit from separability if the approximations are simple enough).

From a loss of separability point of view, augmented Lagrangian (AL) methods or sequential unconstrained minimization techniques (SUMT) do not help. Consider for example the popular augmented Lagrangian statement due to Rockafellar [13]:

$$\tilde{L}_\rho^k(\mathbf{x}, \boldsymbol{\mu}) = \tilde{f}_0^k(\mathbf{x}) + \sum_{j=1}^{m} \mu_j \tilde{f}_j^k(\mathbf{x}) + \frac{\rho}{2} \sum_{j=1}^{m} \left(\tilde{f}_j^k(\mathbf{x}) \right)^2, \tag{4.4}$$

where the μ_j represent the Lagrangian multipliers, the $\tilde{f}_j^k(\mathbf{x})$ the separable constraint approximation functions and ρ a nonzero (positive) penalty parameter (it is often actually common practice to use m penalty parameters ρ_j). For subproblem k, the optimal duo $(\mathbf{x}^{k*}, \boldsymbol{\mu}^{k*})$ is found by iteratively minimizing (4.4) w.r.t. $\mathbf{x}$ with ρ^k and $\boldsymbol{\mu}^k$ fixed; then updating the multipliers $\boldsymbol{\mu}$ and the penalty parameter ρ – the latter often conditionally – until convergence occurs on the subproblem level. The multipliers are updated using the famous Hestenes–Powell formula, e.g. see [39, 40], etc.

The minimization of (4.4) w.r.t. $\mathbf{x}$ may be done using any suitable minimizer that is able to accommodate the bound constraints $\check{\mathbf{x}}$, $\hat{\mathbf{x}}$.

Unfortunately, a disappointing, if obvious feature of (4.4) is that the separable nature of primal approximate subproblem $\mathcal{P}_P[k]$ is not preserved, due to the effects of the squared terms. Penalty-based SUMT suffer from the same drawback.

Enter the so-called *separable* augmented Lagrangian algorithm, or `SALA`, popularized by Hamdi and his co-workers [41, 42, 43, 44, 45], Boyd and his co-workers [46], and many others. The `SALA` framework preserves the separable nature of the subproblems, and the minimization's over the primal variables x_i result in n uncoupled searches, with the obvious feature that this is an *embarrassingly parallel* operation, while the m dual variable updates are also uncoupled. What is more, the separable nature begs the question whether it is possible to find the *primal minimizers in closed-form*. For sub-problems that are simple enough, this is indeed the case; this includes the important class of separable quadratic programs (QPs).

Although the `SALA` paradigm has to the author's knowledge not been applied in structural optimization, they are quite general, and are receiving some attention in the mathematical programming community. A `SALA` may be considered to be an extension of proximal decomposition methods, and derive from the class of splitting algorithms of Douglas and Rachford [47, 48]; they are often known as *alternating directions* type methods.

An immediate word of warning though: although the `SALA` framework seems very attractive given the combination of uncoupled updates of both the primal and dual variables with the dominance of separable approximations in optimal structural design, algorithm `SALA` suffers from sensitivity to a subproblem scaling parameter. To address this, we will herein use an update strategy for this parameter inspired by recent efforts of Lenoir and Mahey [49] and Boyd [46], combined with a function and constraint scaling strategy inspired by the `ALGENCAN` solver [50].

What is more, to take full benefit from the separable nature of algorithm SALA, suitable hardware like general purpose graphical processor units (GPGPUs) should be exploited, but we will not do so herein. Without this option, it is not clear if algorithm SALA will be competitive with the classical dual methods currently so popular in optimal structural design. Nevertheless, application of algorithm SALA to problems in optimal structural design is interesting in its own right, and in addition seems to have educational value.

Our study makes a few salient contributions: Algorithm SALA is free from any line search, and the primal and dual updates are embarrassingly parallel. Indeed, the primal and dual variable updates are available in closed form. The importance of the algorithm using closed-form primal and dual updates cannot be overstated in the parallel context, where search methods can be difficult to implement on a massively parallel scale. Furthermore, the algorithm can easily exploit the use of intervening variables, which are popular and proven in structural optimization.

Our study is arranged as follows: In Section 4.3 we present some diagonal quadratic approximations that are necessary for the separability of SALA. In Section 4.5 we outline the alternating directions type method that we rely upon. We again emphasize that we rely on (strictly) convex approximations in doing so. We then proceed with numerical experiments in Section 4.9, where we compare SALA to the well-known ALGENCAN [50] and the Galahad [51] LSQP solvers, followed by conclusions and recommendations in Section 4.10.

4.3 Some diagonal quadratic approximations

The approximate subproblems used are based on an incomplete series expansion (ISE) [2]. We construct approximations $\tilde{f}_j^k(\mathbf{x})$ to the objective function $f_0(\mathbf{x})$ and all the constraint functions $f_j(\mathbf{x})$ at the point $\mathbf{x}^k$, such that

$$\tilde{f}_j^k(\mathbf{s}) = f_j^k + \boldsymbol{\nabla} f_j^{k\mathrm{T}}\mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}\mathbf{C}_j^k\mathbf{s}, \tag{4.5}$$

with $\mathbf{s} = (\mathbf{x} - \mathbf{x}^k)$ and each $\mathbf{C}_j^k$ an appropriate approximate *diagonal* Hessian matrix, for $j = 0, 1, 2, \ldots, m$. We will occasionally use the abbreviated notation

$$f_j^k = f_j(\mathbf{x}^k),$$

etc. For the sake of clarity, we rewrite (4.5) using summation convention as

$$\tilde{f}_j^k(\mathbf{x}) = f_j^k + \sum_{i=1}^{n}\left(\frac{\partial f_j}{\partial x_i}\right)^k (x_i - x_i^k) + \frac{1}{2}\sum_{i=1}^{n} c_{2i_j}^k (x_i - x_i^k)^2, \tag{4.6}$$

with the $c_{2i_j}^k$ approximate second order diagonal Hessian terms or curvatures. We will herein consider two very simple instances of (4.6), in which the the curvatures are chosen as follows:

1. Such that the approximate function value at the previous iterate $\tilde{f}^{k-1}$ is equal to the real function value f^{k-1} at the previous iterate, being a spherical quadratic approximation (denoted SPH-QDR).

2. Such that the approximation becomes the quadratic approximation to the reciprocal approximation (denoted T2:R), being closely related to the very popular MMA [8] approximations.

While many other possibilities exist, we only outline the above approximations in Appendix 4.4. For the sake of brevity, the reader is referred to [38] and the references therein for details about some other possibilities.

Since the approximations (4.6) are (diagonal) quadratic, primal problem $\mathcal{P}_{NLP}$ may trivially be transformed into a sequence of quadratic approximate programs $\mathcal{P}_{PQ}[k]$, written as

□ *Quadratic approximate program* $\mathcal{P}_{PQ}[k]$

$$\begin{aligned} \min_{\mathbf{s}} \ & \tilde{f}_0^k(\mathbf{s}) = f_0^k + \boldsymbol{\nabla} f_0^{k\mathrm{T}}\mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}\mathbf{Q}^k\mathbf{s} \\ \text{subject to} \ & \tilde{f}_j^k(\mathbf{s}) = f_j^k + \boldsymbol{\nabla} f_j^{k\mathrm{T}}\mathbf{s} = 0, \qquad j = 1, 2, \ldots, m, \\ & \check{x}_i^k \leq x_i \leq \hat{x}_i^k, \qquad i = 1, 2, \ldots, n, \end{aligned} \tag{4.7}$$

with $\mathbf{s} = (\mathbf{x} - \mathbf{x}^k)$ and $\mathbf{Q}^k$ the Hessian matrix of the approximate Lagrangian $\mathcal{L}^k$. For details, the reader is referred to Etman *et al.* [52, 53]. Since the Lagrangian multipliers $\boldsymbol{\mu}^{k*}$ and $\boldsymbol{\lambda}^{k*}$ at the solution of subproblem $\mathcal{P}_{PQ}[k]$ are unknown, the multipliers $\boldsymbol{\mu}^k$ and $\boldsymbol{\lambda}^k$ are used to construct the quadratic program. Hence,

$$Q_{ii}^k = c_{2i_0}^k + \sum_{j=1}^{m} \mu_j^k c_{2i_j}^k, \tag{4.8}$$

and $Q_{id}^k = 0 \ \forall \ i \neq d$, i and $d = 1, 2, \ldots, n$. Note that we do not use slack variables for the inequalities in the Galahad [51] solver LSQP, since doing so would put the said interior-point solver at a disadvantage.

We require the approximate Lagrangian $\mathcal{L}^k$ to be positive definite. Since $\mathbf{Q}^k$ is diagonal, positive definiteness simply requires the individual diagonal elements Q_{ii}^k to be positive. As μ_j^k, c_{2i_0} and c_{2i_j} in principle are unrestricted in sign, we will enforce

$$Q_{ii}^k = \max(\epsilon > 0, Q_{ii}^k), \tag{4.9}$$

with ϵ prescribed and 'small'.

4.4 The approximations used

4.4.1 A spherical diagonal quadratic approximation (SPH-QDR)

To construct a spherical quadratic approximation [54], we select $c_{2i_j}^k \equiv c_{2_j}^k \ \forall \ i$, which requires the determination of the single unknown $c_{2_j}^k$, to be obtained by (for example) enforcing the condition

$$\tilde{f}_j(\mathbf{x}^{k-1}) = f_j(\mathbf{x}^{k-1}), \tag{4.10}$$

which implies that

$$c_{2_j}^k = \frac{2[f_j(\mathbf{x}^{k-1}) - f_j(\mathbf{x}^k) - \boldsymbol{\nabla} f_j^{k\mathrm{T}}(\mathbf{x}^{k-1} - \mathbf{x}^k)]}{\|\mathbf{x}^{k-1} - \mathbf{x}^k\|_2^2}. \tag{4.11}$$

This results in the approximation proposed by Snyman and Hay [54]. For the first iteration, when no historic information is available, curvatures of unity are assumed. An alternative condition for formulating a spherical quadratic approximation is presented in Reference [55].

4.4.2 The quadratic approximation to the reciprocal approximation (T2:R)

For reciprocal intervening variables, the second order partial derivatives $c_{2i_j}^k$ are obtained [38] as

$$c_{2i_j}^k = \frac{\partial^2 \tilde{f}_{\mathrm{R}}}{\partial x_i^2}\left(\mathbf{x}^k\right) = \frac{-2}{x_i^k}\left(\frac{\partial f_j}{\partial x_i}\right)^k, \tag{4.12}$$

where $\tilde{f}_{\mathrm{R}}$ indicates the reciprocal approximation; see also [56]. Of course, (4.12) is only sensible if $\partial f_j/\partial {x_i}^k < 0$. If not, we choose to use

$$c_{2i_j}^k = \left|\frac{\partial^2 \tilde{f}_{\mathrm{R}}}{\partial x_i^2}\left(\mathbf{x}^k\right)\right| = \frac{2}{x_i^k}\left(\frac{\partial f_j}{\partial x_i}\right)^k, \tag{4.13}$$

which admittedly is very conservative, but this choice has served us well previously. Nevertheless, many other possibilities exist.

4.5 Alternating directions type methods (ADMM)

We proceed with a brief outline of the salient features of algorithm `SALA`; for details, the reader is referred to the cited literature in the mathematical programming community[2]. Here, we follow closely the presentation of Lenoir and Mahey [49]. In essence, algorithm `SALA` reformulates separable problem (4.3) with the help of the allocation subspace

$$\mathcal{A} = \left\{ y_{ji} \in \mathcal{R}^{mn} / \sum_{i=1}^{n} y_{ji} = 0, j = 1, 2, \ldots, m \right\}. \tag{4.14}$$

Then, the allocation of resource vectors $y_{ji} = \tilde{f}_{ji}^k(x_i)$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m$ results in an embedded formulation of problem (4.3) with a distributed coupling, written as

$$\begin{aligned} \min_{\mathbf{x}} \quad & \sum_{i=1}^{n} \tilde{f}_0^k(x_i) \\ \text{subject to} \quad & y_{ji} = \tilde{f}_{ji}^k(x_i), \\ & y_{ji} \in \mathcal{A}, \\ & \check{x}_i^k \le x_i \le \hat{x}_i^k, \end{aligned} \tag{4.15}$$

[2] An interesting recent application of ADMM in structural optimization may be found in [57].

for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m$.

The *approximate* augmented Lagrangian $\mathcal{L}_\rho$ with penalty parameter $\rho > 0$ obtained by associating the multipliers μ_{ji} with the allocation constraints $y_{ji} = \tilde{f}_{ji}^k(x_i)$ decomposes into the sum

$$\tilde{\mathcal{L}}_\rho^k(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) = \sum_{i=1}^{n} \tilde{\mathcal{L}}_{\rho,i}^k(x_i, y_{ji}, \mu_{ji}), \tag{4.16}$$

with

$$\tilde{\mathcal{L}}_{\rho,i}^k(x_i, y_{ji}, \mu_{ji}) = \tilde{f}_0^k(x_i) + \sum_{j=1}^{m} \mu_{ji}(\tilde{f}_{ji}^k(x_i) - y_{ji}) + \frac{\rho}{2} \sum_{j=1}^{m} (\tilde{f}_{ji}^k(x_i) - y_{ji})^2. \tag{4.17}$$

The stationary point of $\tilde{\mathcal{L}}_{\rho,i}^k(x_i, y_{ji}, \mu_{ji})$ is obtained via successive minimization's over the x_i and the y_{ji} in a Gauss-Seidel fashion as to exploit the separability of (4.15).

The minimization's in the x_i yield the n independent subproblems

$$\min_{x_i} \tilde{\mathcal{L}}_{\rho,i}^k(x_i, y_{ji}^k, \mu_{ji}^k),$$

which can be done in parallel. Since the μ_{ji} are in $\mathcal{A}^\perp$, with $\mathcal{A}$ and $\mathcal{A}^\perp$ mutually orthogonal, we obtain the updates for the y_{ji} as

$$y_{ji}^{l+1} = \tilde{f}_{ji}^k(x_i^{l+1}) - \frac{1}{n}\left(\tilde{f}_j^k(x_i^{l+1})\right), \tag{4.18}$$

again see Lenoir and Mahey, and Boyd for details and proofs. Subspace $\mathcal{A}^\perp$ has the explicit formulation

$$\mathcal{A}^\perp = \{\mu_{ji} \in \mathcal{R}^{mn} / \mu_{j1} = \mu_{j2} = \ldots = \mu_{jn}, j = 1, 2, \ldots, m\}. \tag{4.19}$$

So, at every iteration l, knowledge of the μ_{ji} reduces to the knowledge of its common component $\nu_j = \mu_{j1} = \mu_{j2} = \ldots = \mu_{jn}, j = 1, 2, \ldots, m$, and the update step for the ν_j becomes

$$\nu_j^{l+1} = \nu_j^l + \frac{\rho}{n}\left(\tilde{f}_j^k(x_i^{l+1})\right), \tag{4.20}$$

again see the cited literature for details. The complete resulting algorithm `SALA`, without the trivial objective function and constraint scaling given in (4.25), is listed in Algorithm 1. Again note that not only are the n uncoupled minimization's over the x_i embarrassingly parallel; updating the y_{ji} and ν_j is also embarrassingly parallel.

We impose a subproblem convergence criteria on both the step sizes made by the primal and dual variables, as well as the maximum number of subproblem evaluations.

4.6 Closed-form expressions for QP-like problems

If the functions $\tilde{f}_{ji}^k$ are simple enough, the n one-dimensional minimization's over the x_i may even be done in closed-form. For a separable QP-like subproblem, this is indeed the case. Let

$$\tilde{f}_0^k(x_i) = a + b_i(x_i - x_i^k) + \frac{1}{2} c_i (x_i - x_i^k)^2, \tag{4.21}$$

Algorithm 1: Algorithm `SALA` for a *given* equality constrained *sub*problem k

Initialize: $l = 0$, $\epsilon > 0$, $\rho^0 > 0$, $y_{ji}^0 \in \mathcal{A}$, $\nu_j^0 = \in \mathcal{A}^\perp$
repeat
  for $i = 1, n$ **do**
    $x_i^{l+1} \leftarrow \arg\min_{x_i} \tilde{\mathcal{L}}_{\rho,i}^k(x_i, y_{ji}^l, \nu_j^l)$
  end
  for $j = 1, 2, \ldots, m$ **and** $i = 1, 2, \ldots, n$ **do**
    update y_{ji}^{l+1} using (4.18)
  end
  for $j = 1, 2, \ldots, m$ **do**
    update ν_j^{l+1} using (4.20)
  end
  update ρ^{l+1} using (4.30)
  $l \leftarrow l + 1$
until $\|\boldsymbol{\nu}^{l+1} - \boldsymbol{\nu}^l\| \leq \epsilon$ **and** $\|\mathbf{x}^{l+1} - \mathbf{x}^l\| \leq \epsilon$ **or** $l \leq l_{max}$
end

with $\mathbf{b}$ and $\mathbf{c}$ given n-vectors, and

$$\tilde{f}_j^k(x_i) = d_{ji} + e_{ji}(x_i - x_i^k), \tag{4.22}$$

again with $\mathbf{d}_j$ and $\mathbf{e}_j$ given n-vectors. In order to satisfy (4.3), we have chosen to distribute the j constraint values equally among each of their n separable functions such that

$$d_{ji} = f_j(\mathbf{x}^k)/n. \tag{4.23}$$

We have now resorted to summation convention – a sum over repeated indices in a term is implied, i.e. we sum over i in (4.21) and (4.22) above. Here, the c_i represent curvature information of the objective and the constraint functions, see Etman *et al.* [52, 53], and we assume *strictly convex* primal approximate subproblems $\mathcal{P}_P[k]$. Then, the stationary conditions of (4.17) w.r.t. the x_i result in

$$x_i^{l+1} = \Pi\left(x_i^k - \left(c_i + \rho e_{ji}^2\right)^{-1}\left(b_i + \nu_j e_{ji} - \rho y_{ji} e_{ji} + \rho d_{ji} e_{ji}\right)\right), \tag{4.24}$$

and we sum over j, with $\Pi(\cdot)$ the projection onto $\left[\check{x}_i^k, \hat{x}_i^k\right]$. Note that the denominator cannot vanish, since $c_i > 0$ and $\rho > 0$ (although $e_{ji} = 0$ is possible in sparse problems). Line 4 in Algorithm 1 may thus be replaced by the very simple closed-form expression (4.24). The relation between (4.6) and (4.21), (4.22) is clear; also see [52, 53].

As mentioned, suitable separable approximations are briefly presented in Section 4.3. For the sake of clarity, we here again mention that we use a spherical quadratic approximation (denoted SPH-QDR), and a quadratic approximation to the reciprocal approximation (denoted T2:R), being closely related to the very popular MMA [8] approximations. For some examples which are truly separable, we also use exact Hessian information.

4.7 Objective and constraint function scaling

The SALA algorithm is highly sensitive to the scaling in the objective function, constraints and subproblem. We therefore scale both the objective and constraint functions, such that

$$\begin{aligned} \min_{\mathbf{x}} \ & w_f f_0(\mathbf{x}) \\ \text{subject to} \ & w_{c_j} f_j(\mathbf{x}) = 0, \qquad j = 1, 2, \ldots, m, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, \qquad i = 1, 2, \ldots, n, \end{aligned} \tag{4.25}$$

where the scaling factors, w_f and w_{c_j}, are given by

$$\begin{aligned} w_f &= 1/\max(\|\boldsymbol{\nabla} f_0(\mathbf{x}^{k-1})\|_\infty, 1), \\ w_{c_j} &= 1/\max(\|\boldsymbol{\nabla} f_j(\mathbf{x}^{k-1})\|_\infty, 1), \qquad j = 1, 2, \ldots, m, \end{aligned}$$

with $\mathbf{x}^k$ denoting the primal values for subproblem k. Throughout our numerical testing, the proposed scaling strategy ensured a satisfactory subproblem trade-off between convergence rate and conditioning. Different scaling strategies, such us using the Euclidean norm, were not as effective as (4.25) over a representative test set.

4.8 Subproblem scaling

The projected subgradient step given in (4.20) depends on the step length ρ, with this parameter behaving more like a scaling parameter in the SALA framework than the penalty parameter in classical augmented Lagrangian statements. For example, ρ penalizes the primal coupling constraints, and greater values will accelerate the primal sequence. However, ρ^{-1} penalizes the dual sequence, so that a compromise value is expected to be optimal, e.g. see Lenoir and Mahey [49], Boyd [46], who elaborate on optimal scaling strategies in some detail. It is possible to use ρ_i for $i = 1, 2, \ldots, n$ scaling parameters for each of the separable subproblems; but this only leads to increased computational costs [49]. Furthermore, if the objective and constraint functions are scaled, the n separable problems have similar orders of magnitude, which further reduces the need for individual separable subproblem scaling parameters. The single parameter update is therefore the preferred choice of subproblem scaling for our numerical testing.

We obtained reasonable numerical results when using a combination of the strategies inspired by Lenoir and Mahey [49] and Boyd [46].

From [49] , let

$$\beta = \frac{\|\boldsymbol{\nu}^l - \boldsymbol{\nu}^{l-1}\|}{\|\mathbf{y}^l - \mathbf{y}^{l-1}\|}. \tag{4.26}$$

Then, we update using

$$\rho \leftarrow \rho^{(1-\alpha)} + \beta^\alpha, \tag{4.27}$$

or

$$\rho \leftarrow (1 - \alpha)\rho + \alpha\beta, \tag{4.28}$$

with α small, say 10^{-2} , to prevent oscillatory behavior. We accept that Lenoir and Mahey [49] prove that

$$\sum_{l=1}^{\infty} \alpha_l = S < \infty \tag{4.29}$$

is a requirement for theoretical convergence, but for numerical purposes $l \ll \infty$, resulting in small values of α still achieving practical problem convergence.

We now introduce the update strategy inspired by [46], coupled with that from Equation (4.28). The update then becomes

$$\rho \leftarrow \begin{cases} \sqrt{n}\rho & \text{if } \|\mathbf{x}^l - \mathbf{x}^{l-1}\| > n\|\boldsymbol{\nu}^l - \boldsymbol{\nu}^{l-1}\| \\ \rho/\sqrt{n} & \text{if } \|\boldsymbol{\nu}^l - \boldsymbol{\nu}^{l-1}\| > n\|\mathbf{x}^l - \mathbf{x}^{l-1}\| \\ (4.27) \text{ or } (4.28) & \text{otherwise.} \end{cases} \tag{4.30}$$

The update strategy proposed by (4.27) and (4.28) takes advantage of second-order problem information [49], which in our numerical testing led to higher levels of accuracy compared to a strategy that purely focused on keeping the primal and dual residual norms within a factor of each other. The downside, however, is the possibility of ρ quickly jumping to unsuitably high values should the resource allocation vector and dual residual norms be of different orders of magnitude. Furthermore, it is difficult to reduce the value of the scaling parameter because of the low values of α that have to be chosen in order to ensure convergence. We combine the two different strategies in the hope of ensuring relatively equal convergence of both the primal and dual sequences, with the advantage of greater levels of accuracy.

4.9 Numerical experiments

We use the QP form given in (4.6) in Section 4.3 to construct an approximate augmented Lagrangian for both the `ALGENCAN` and `SALA` solvers. For `SALA`, the augmented Lagrangian is then decomposed further into the n separable Lagrangian problems given in (4.17). We reiterate that the approximations used for the Hessian are either the spherical diagonal quadratic approximation (SPH-QDR) or the quadratic approximation to the reciprocal approximation (T2:R). We also compare our algorithms with the state-of-the-art Galahad [51] solver `LSQP`. The latter solver is tailor-made for linear or *diagonal* quadratic subproblems.

All numerical results for `SALA` use the closed-form update of the primal variables given in (4.24). The step length ρ is updated using (4.28) and (4.30) with $\alpha = 10^{-2}$. The maximum number of subproblem evaluations allowed is $l_{max} = 5 \times 10^4$ with a subproblem convergence tolerance of $\epsilon = 10^{-6}$. For each subproblem k, we initialize at $l = 0$ as follows: For values of $k > 0$, $\rho^0 = 1$, $y_{ji}^0 = y_{ji}^{k-1}$ and $\nu_j^0 = \nu_j^{k-1}$. Else, at $k = 0$, $\mathbf{y} = \boldsymbol{\nu} = \mathbf{0}$. A maximum of $k = 500$ outer iterations was allowed, after which the algorithm was stopped, and we indicate failure to converge by '—' in the numerical results to follow. The absolute values of the Lagrangian multipliers were bounded to not exceed 10^6, selected rather arbitrarily, to prevent numerical instabilities.

Default settings are used for the `ALGENCAN` solver, except that we use $epsfeas = epsopt = 10^{-7}$, where *epsfeas* and *epsopt* are parameters in `ALGENCAN`[3]. For Galahad's `LSQP` solver, default settings were used.

We now introduce the absolute maximum constraint violation h, the norm of the KKT conditions $\mathcal{K}$, and the number of function evaluations N_f required for termination. For all the solvers, problem execution was terminated when all of three conditions were met, namely the Euclidean or 2-norm $\|\mathbf{x}^k - \mathbf{x}^{k-1}\|_2 \leq 10^{-4}$, $h \leq 10^{-4}$, and $\mathcal{K} \leq 10^{-3}$. We chose convergence tolerances of modest accuracy, as ADMM is usually slow to converge to high accuracy [46]. The test problems used are tabulated in Table 4.1, and we present results in Table 4.2.

Table 4.1: The test problems. 'Approx.' denotes the Hessian approximation used. Note 1: This is a generalization of Svanberg's cantilever. Note 2: The problem is separable. Note 3: The Hock and Schittkowski test set.

PR	Problem name	Approx.	n	m	Ref.	Notes
1	Svanberg's cantilever	SPH-QDR	5	1	[8]	
2	Toropov's cantilever	T2:R	1024	1	[11, 58]	1
3	Svanberg's first non-convex problem	SPH-QDR	200	2	[59]	
4	Svanberg's second non-convex problem	SPH-QDR	200	2	[59]	
5	12-Corner-polytope problem	SPH-QDR	21	1	[9]	
6	Vanderplaats' cantilever #1	T2:R	200	201	[60]	
7	Vanderplaats' cantilever #2	T2:R	200	200	[60]	
8	Vanderplaats' cantilever #3	T2:R	200	200	[60]	
9	Fleury's weight minimization problem	T2:R	1000	2	[61]	
10	Svanberg's snake problem	SPH-QDR	30	41	[62]	
11	Cam-design problem	SPH-QDR	15	49	[63]	
12	Svanberg's cantilever	Exact	5	1	[8]	2
13	HS-43	Exact	4	3	[64]	2, 3

The results are not overly surprising; sometimes `ALGENCAN` performs slightly better than `SALA`, and vice versa. This happens notwithstanding the fact that the two algorithms solve a sequence of identical subproblems, reminiscent of the no-free-lunch (NFL) theorems of optimization [65]. Arguably, the main reason for this is in all probability that *both* algorithms are known to be sensitive to scaling, albeit that augmented Lagrangian methods are probably less so than ADMM type methods. Arguably, Galahad's `LSQP` solver sets the tone. (Note that $\mathcal{K}$ is smaller for `LSQP` than for the other solvers; notwithstanding that the same stopping criteria were used.)

The results for Svanberg's snake problem are worthy of special attention: for this problem, none of the algorithms converged. We have on purpose included these results to highlight

[3]We found these optimality tolerances to be sufficient for solving the subproblems, with stricter tolerances resulting in a negligibly different iteration path.

some limitations of the methods studied. It is pertinent to point out that this problem is very 'difficult'. The snake problem consists of a very thin feasible slither in n-dimensional space. Is is also of note to mention that arguably the most popular algorithms in structural optimization, namely CONLIN and MMA, also have difficulties solving some of the test problems. CONLIN for example fails when applied to the snake problem, whereas MMA fails for Fleury's weight minimization problem; all in the spirit of NFL.

4.10 Conclusions and recommendations

We have proposed an iterative separable augmented Lagrangian algorithm (`SALA`) for optimal structural design. The algorithm solves a sequence of separable quadratic-like programs, able to capture reciprocal and exponential-like behavior, which is desirable in structural optimization. A salient feature of the algorithm is that the primal and dual variable updates are all updated using closed-form expressions.

For further reading, the reader is referred to the monograph by Boyd *et al.* [46], and some of the references mentioned therein, for instance Bertsekas. The first five chapters of the monograph by Boyd *et al.* give an overview of the main ADMM concepts and methods, and they also treat the special case of the QP and separable objective and constraints.

Since algorithms like `SALA` are known to be very sensitive to scaling, we have proposed a scaling method inspired by the well-known `ALGENCAN` algorithm. Numerical results for `SALA` and `ALGENCAN` suggest that the algorithms perform quite similar. Having said this: "optimal" scaling of the algorithm (and related algorithms) remains an open issue.

Indeed, the sensitivity of `SALA` to scaling and different parameter settings may well be its biggest drawback. Having said this: even 'classical' AL statements well known to be prone to this, and the same even applies for penalty methods. To take this argument even further: 'older' SQP formulations that depended on a merit acceptance function revealed the same difficulties.

The scaling issue posed by `SALA` is of such complexity, that Lenoir and Mahey [49] dedicated an entire study to this issue. Interested readers may find an in depth analysis of the effect of different scaling strategies therein. As mentioned before, scaling is an open ended problem that will require further investigation, but is beyond the scope of this study.

A feature we exploit in Section 4.12.2, the primal and dual updates in `SALA` are both embarrassingly parallel, which makes the algorithm suitable for implementation on massively parallel computational devices, including general purpose graphical processor units (GPGPUs). For some problems, a relatively slow ADMM iteration process leads to increased computational effort on the subproblem level, when compared to more traditional SQP methods. We noted this in our numerical testing, as `ALGENCAN` and `LSQP` sometimes required less CPU time than a single threaded `SALA` implementation. However, `SALA` is theoretically divisible in problem time by at most n, meaning that the advantage over traditional SQP methods should scale with increased problem size. (Problem dimensionality will of course also have to be high enough to offset the overhead computational costs associated with parallel computing.)

Finally, while we have used only two approximations herein, many a different method for obtaining the diagonal approximate higher order curvatures may be used. Possibilities include quasi-Cauchy-updates [66], and quadratic approximations to suitable intervening variables [38, 56], etc. The last possibility includes the quadratic approximation to the reciprocal and exponential approximations, and even the quadratic approximations to the CONLIN and MMA approximations.

4.11 Tables for numerical results

Table 4.2: Numerical results for the test problems using the ALGENCAN and SALA solvers. Superscript * indicates the final values at termination, while '—' indicates that the algorithm failed to terminate.

	ALGENCAN				SALA				LSQP			
PR	f_0^*	h^*	$\mathcal{K}^*$	N_f	f_0^*	h^*	$\mathcal{K}^*$	N_f	f_0^*	h^*	$\mathcal{K}^*$	N_f
1	1.3399564	1.27×10^{-8}	2.89×10^{-5}	13	1.3399564	0.00×10^{0}	8.43×10^{-6}	20	1.3399563	6.48×10^{-8}	2.89×10^{-5}	13
2	1.3103299	3.04×10^{-7}	3.63×10^{-8}	8	1.3103281	4.46×10^{-6}	3.32×10^{-13}	8	1.3103301	0.00×10^{0}	7.75×10^{-13}	8
3	51.046453	0.00×10^{0}	1.02×10^{-2}	100	51.046524	4.31×10^{-4}	1.17×10^{-2}	97	51.046253	0.00×10^{0}	8.49×10^{-3}	104
4	-148.95382	2.61×10^{-6}	7.02×10^{-3}	164	-148.95388	2.71×10^{-4}	6.08×10^{-3}	169	-148.95382	2.07×10^{-6}	7.02×10^{-3}	164
5	-279.90266	2.01×10^{-7}	8.15×10^{-3}	136	-279.90246	0.00×10^{0}	7.28×10^{-3}	126	-279.90215	5.67×10^{-8}	5.82×10^{-3}	146
6	63678.094	1.90×10^{-7}	7.49×10^{-3}	9	63678.897	8.07×10^{-5}	3.98×10^{-3}	9	63678.100	3.33×10^{-10}	7.50×10^{-3}	9
7	54176.212	5.53×10^{-9}	2.83×10^{-5}	8	54176.190	4.06×10^{-6}	8.33×10^{-4}	8	54176.212	1.07×10^{-13}	1.98×10^{-5}	8
8	54155.571	3.31×10^{-7}	1.24×10^{-3}	8	54155.570	1.68×10^{-6}	4.67×10^{-2}	8	54155.571	1.92×10^{-8}	1.24×10^{-3}	8
9	950.00005	7.93×10^{-7}	7.75×10^{-5}	22	950.00136	0.00×10^{0}	2.79×10^{-3}	20	950.00003	1.52×10^{-5}	7.78×10^{-3}	32
10	—	—	—	—	—	—	—	—	—	—	—	—
11	-4.3452690	1.14×10^{-7}	8.65×10^{-5}	5	-4.3815720	3.36×10^{-4}	8.12×10^{-4}	32	-4.3452583	3.55×10^{-15}	1.79×10^{-6}	6
12	1.3399566	0.00×10^{0}	3.37×10^{-7}	13	1.3399564	3.12×10^{-9}	1.66×10^{-5}	15	1.3399563	3.56×10^{-8}	1.47×10^{-8}	13
13	-44.000002	2.05×10^{-6}	1.10×10^{-5}	6	-44.000003	2.47×10^{-6}	3.04×10^{-5}	8	-44.000003	1.17×10^{-6}	1.15×10^{-5}	6

4.12 Further work

Since this chapter follows the layout from our original study in [35], we present our additional work as a separate section herein.

4.12.1 Exploiting constraint Jacobian sparsity

The strong coupling between the dual variables μ_j, distributed constraints d_{ji} and resource allocation vectors y_{ji} with the constraint Jacobian terms in (4.24), denoted by e_{ji}, is clear. For a sparse Jacobian, our initial separable constraint formulation in (4.23) results in sub-optimal constraint distribution, since the separable constraint terms are absorbed by the zero e_{ji} terms. Additionally, the resource allocation vector terms associated with the zero e_{ji} terms become redundant.

Perhaps a more sensible approach would be to distribute all separable terms over only the nonzero e_{ji} terms, which still preserves the definition in (4.15). Let the cardinality, or number of nonzero elements, of the first-order constraint vector $\mathbf{e}_j$ be denoted by ζ_j. The constraints can then be uniformly distributed over their respective nonzero Jacobian entries, such that

$$d_{ji} = fx_j(\mathbf{x}^k)/\zeta_j, \quad i = 1, \ldots, \zeta_j. \tag{4.31}$$

The resource allocation vectors and slack variables can now trivially be reformulated to be coupled with our separable constraint terms in (4.31), breaking the coupling any of the variables have with the zero e_{ji} terms.

Optimizing the constraint and resource allocation vector distributions results in significantly less subproblem evaluations, with the results for single-threaded CPU implementations presented in Table 4.3. The three Vanderplaats' problems in [60] are chosen for their sparse properties, using the identical solver settings and result notation as in Section 4.9. An additional column N_l is added to denote the subproblem evaluations required.

Table 4.3 shows that the sparse implementation requires substantially less subproblem evaluations, while retaining the salient features of the dense implementation. Serial implementations of `SALA` will still benefit from a reduction in solver solution time, since in addition to the fewer subproblem evaluations, sparse linear algebra may be exploited. Unfortunately, Vanderplaats' #1 did not converge for the sparse implementation; in high probability due to the problem formulation requiring dual variables of massively differing orders of magnitude. It is possible that the sparse implementation is more sensitive to scaling, although further numerical testing is warranted to confirm this.

4.12.2 GPGPU implementation

As previously mentioned, `SALA` has the salient feature of embarrassingly parallel primal and dual variable updates, which we attempt to exploit herein. Although many possibilities exist to exploit the structure of multi-core CPUs and GPGPUs, we will focus on OpenMP [67] and NVIDIA's CUDA [68] respectively. Both of the aforementioned solutions provide native support for FORTRAN, which is the language of choice for our SAOi framework [3], where

SALA resides. Custom OpenMP and CUDA kernels were written for the primal updates in (4.24) and the dual updates in (4.20), as well as for the embarrassingly parallel resource allocation vector updates in (4.18).

To generate the numerical results, we used the test platform from Section 6.7, together with the solver settings in Section 4.9. The serial and OpenMP results are compiled using gfortran-9, while the GPGPU results use the PGI FORTRAN compiler detailed in Section 6.7. It must be noted that our OpenMP implementation was limited to using four threads on our eight thread (quad-core) Xeon processor. We strictly focused on large-scale variations of Vanderplaats' #2 and #3[4] [60], due to the problems scaling in terms of primal variables and constraints. Unfortunately, sparse SALA did not converge for Vanderplaats' #1, henceforth we only include results for Vanderplaats' #2 and #3.

We compare our *sparse* SALA implementations to the state-of-the-art Galahad [51] LSQP solver, which as previously mentioned, is tailor-made for linear or *diagonal* quadratic subproblems. The results of the ALGENCAN solver were not included, since the large-scale problem solution times were prohibitively expensive. It must be noted that our custom CUDA and OpenMP kernels leave plenty of room for optimization, both in terms of memory access patterns as well as for targeting specific hardware architectures. Furthermore, we implemented all solver routines within our existing SAO*i* framework [3], thereby introducing GPGPU data-transfer overheads that we *include* in our subproblem solver times. The SALA serial (and LSQP), OpenMP and GPGPU solver times are measured in seconds and denoted by t, t_{OMP} and t_{GPU} respectively. Finally, the numerical results are presented in Table 4.4.

Figures 4.1 and 4.2, along with Table 4.4, show the significant reduction in solution time when exploiting the separability prevalent in SALA. The OpenMP implementation of SALA requires less solution time than the serial implementation, while the GPGPU implementation in turn requires significantly less solution time compared to the OpenMP implementation. The GPGPU implementation, despite the penalty of overheads, reduces the serial solution time by over a factor of thirty when $n = m = 10^7$. Unfortunately, technical issues prevented us from testing the OpenMP implementation for $n = m = 10^7$; however, Figures 4.1 and 4.2 do provide insight into the expected solution time for $n = m = 10^7$.

[4] At the solution point, approximately $m/2 = n/2$ constraints are active for both problems.

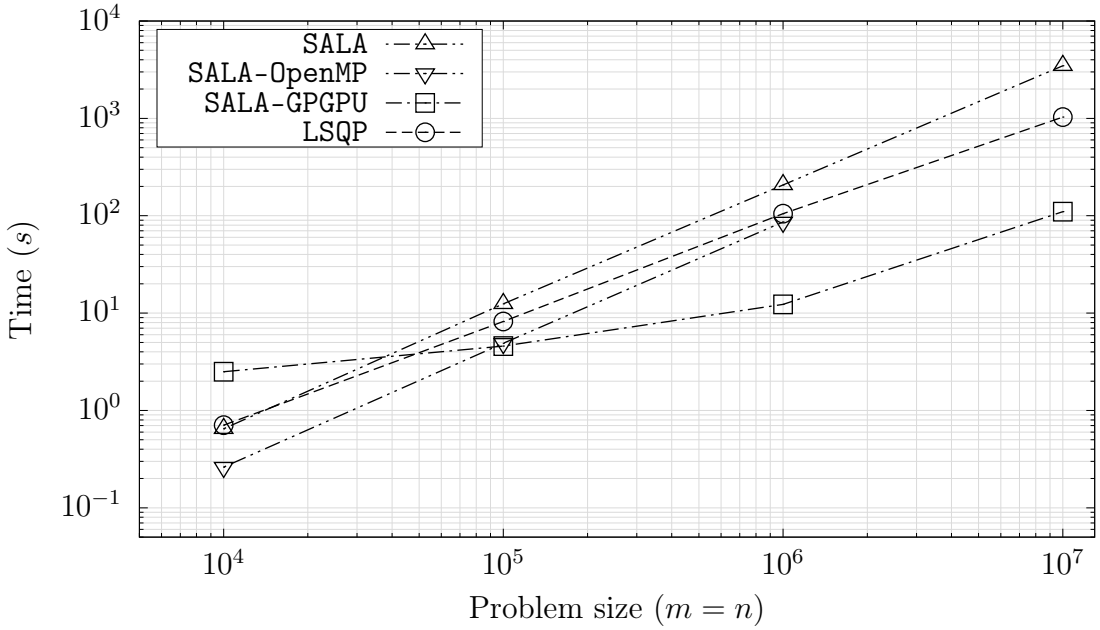


Figure 4.1: The OpenMP and GPGPU implementations of `SALA` for Vanderplaats' #2.

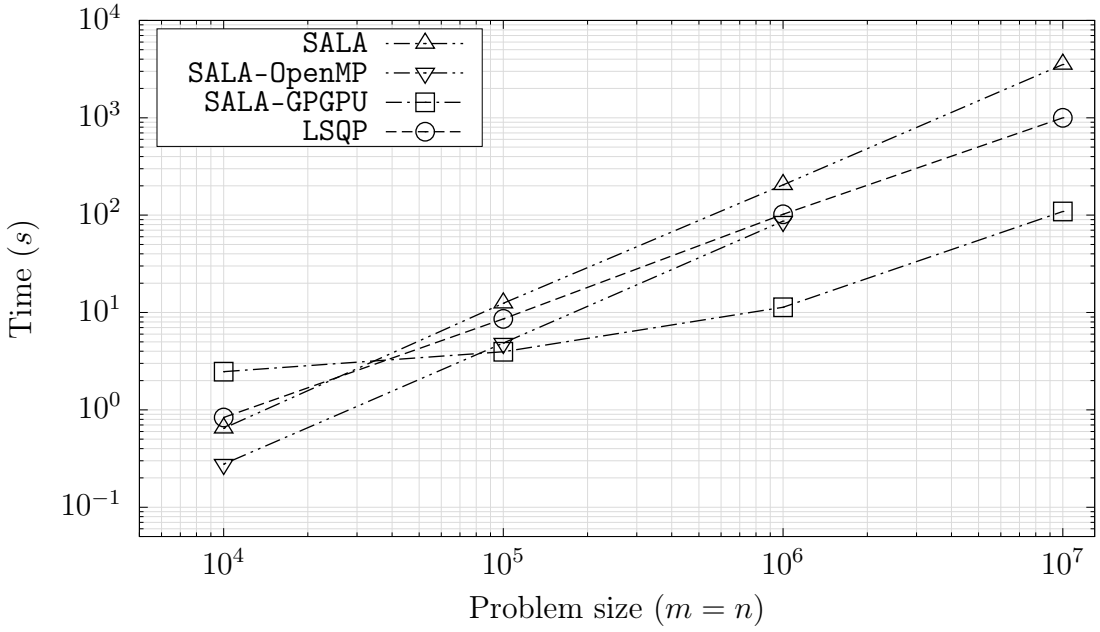


Figure 4.2: The OpenMP and GPGPU implementations of `SALA` for Vanderplaats' #3.

Table 4.3: Numerical results for the Vanderplaats' test problems [60] using the dense and sparse SALA solvers. Superscript * indicates the final values at termination, while '—' indicates that the algorithm failed to terminate.

			Dense					Sparse				
PR	n	m	f_0^*	h^*	$\mathcal{K}^*$	N_f	N_l	f_0^*	h^*	$\mathcal{K}^*$	N_f	N_l
6	300	301	6.367×10^{4}	1.548×10^{-5}	7.020×10^{-4}	12	316427	—	—	—	—	—
	400	401	6.367×10^{4}	6.176×10^{-5}	2.203×10^{-4}	12	353566	—	—	—	—	—
7	300	300	5.403×10^{4}	7.353×10^{-5}	5.191×10^{-4}	27	68954	5.403×10^{4}	4.083×10^{-7}	1.994×10^{-6}	9	858
	400	400	5.396×10^{4}	6.083×10^{-5}	5.545×10^{-4}	25	92169	5.396×10^{4}	3.927×10^{-7}	3.358×10^{-6}	9	1032
8	300	300	5.401×10^{4}	3.561×10^{-5}	9.208×10^{-4}	10	71401	5.401×10^{4}	2.883×10^{-7}	1.989×10^{-4}	9	837
	400	400	5.394×10^{4}	4.548×10^{-5}	2.441×10^{-3}	9	90474	5.394×10^{4}	9.860×10^{-6}	1.205×10^{-4}	9	848

Table 4.4: Comparison of the serial, OpenMP and GPGPU SALA implementations for the Vanderplaats' test problems [60]. Superscript * indicates the final values at termination, while ‡ denotes that a solver failed to start.

		SALA							LSQP				
PR	$n=m$	f_0^*	h^*	$\mathcal{K}^*$	N_f	t (s)	t_{OMP} (s)	t_{GPU} (s)	f_0^*	h^*	$\mathcal{K}^*$	N_f	t (s)
7	10^4	5.372×10^{4}	2.638×10^{-7}	6.424×10^{-6}	10	6.48×10^{-1}	2.62×10^{-1}	2.50×10^{0}	5.372×10^{4}	8.468×10^{-9}	7.672×10^{-6}	10	7.07×10^{-1}
	10^5	5.371×10^{4}	9.264×10^{-5}	7.893×10^{-5}	10	1.25×10^{1}	4.89×10^{0}	4.59×10^{0}	5.371×10^{4}	9.237×10^{-5}	4.234×10^{-5}	10	8.19×10^{0}
	10^6	5.371×10^{4}	4.550×10^{-5}	1.187×10^{-6}	12	2.07×10^{2}	8.63×10^{1}	1.23×10^{1}	5.371×10^{4}	5.463×10^{-5}	1.730×10^{-6}	12	1.05×10^{2}
	10^7	5.371×10^{4}	1.649×10^{-5}	2.943×10^{-8}	13	3.47×10^{3}	‡	1.10×10^{2}	5.371×10^{4}	0.000×10^{0}	1.188×10^{-5}	12	1.03×10^{3}
8	10^4	5.372×10^{4}	2.638×10^{-7}	6.424×10^{-6}	10	6.46×10^{-1}	2.77×10^{-1}	2.47×10^{0}	5.372×10^{4}	8.468×10^{-9}	7.672×10^{-6}	10	8.36×10^{-1}
	10^5	5.371×10^{4}	9.264×10^{-5}	7.893×10^{-5}	10	1.24×10^{1}	4.86×10^{0}	3.95×10^{0}	5.371×10^{4}	9.237×10^{-5}	4.234×10^{-5}	10	8.60×10^{0}
	10^6	5.371×10^{4}	4.550×10^{-5}	1.187×10^{-6}	12	2.04×10^{2}	8.69×10^{1}	1.13×10^{1}	5.371×10^{4}	5.463×10^{-5}	1.730×10^{-6}	12	1.02×10^{2}
	10^7	5.371×10^{4}	1.649×10^{-5}	2.943×10^{-8}	13	3.52×10^{3}	‡	1.09×10^{2}	5.371×10^{4}	0.000×10^{0}	1.188×10^{-5}	12	1.00×10^{3}

Chapter 5

A separable primal-dual algorithm

The work presented here originates from a paper titled "A separable primal-dual algorithm for very large-scale optimal structural design" [69], which is under review at the time of writing. The paper is co-authored by Prof. Albert A. Groenwold.

5.1 Abstract

We propose an efficient separable Lagrangian algorithm (SLA) for optimal structural design, with the salient feature of closed-form updates for both the primal and dual variables. `SLA` solves a sequence of separable quadratic-like programs, able to capture both direct and intervening variables, the latter being popular in structural optimization. For a quadratic-like problem with n design variables and m_a active constraints, the dual variables are found from the solution of an $m_a \times m_a$ positive-definite linear system, making `SLA` ideally suited to problems where $m_a \ll n$. The primal updates are performed in an embarrassingly parallel fashion, requiring a single matrix-vector operation of $\mathcal{O}(nm_a)$. Both the primal and dual variable updates are ideal for implementation on massively parallel computational devices, such as general purpose graphical compute units (GPGPUs), although we do not demonstrate said implementation herein.

To demonstrate the efficiency and scalability of `SLA`, we present results for very large-scale structural problems containing hundreds of millions of design variables and hundreds of millions of constraints. The solutions for these problems require only a few minutes on a single quad-core CPU with 128 GB of RAM. Numerical results are further compared to the state-of-the-art Galahad `LSQP` solver and the popular Falk dual, where `SLA` outperforms both solvers by orders of magnitude with respect to problem solution time.

5.2 Introduction

Consider the constrained nonlinear problem $\mathcal{P}_{\text{NLP}}$, given by

$$\begin{aligned} \min_{\mathbf{x}} \ & f_0(\mathbf{x}) \\ \text{subject to} \ & f_j(\mathbf{x}) = 0, \qquad j = 1, \ldots, m', \\ & f_j(\mathbf{x}) \leq 0, \qquad j = m'+1, \ldots, m, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, \qquad i = 1, \ldots, n, \end{aligned} \tag{5.1}$$

where $f_0(\mathbf{x}) : \mathcal{R}^n \to \mathcal{R}$ is the objective function and $f_j(\mathbf{x}) : \mathcal{R}^n \to \mathcal{R}$, $j = 1, \ldots, m$ are the constraint functions, which depend on the n primal variables $\mathbf{x} \in \mathcal{R}^n$ and are at least once continuously differentiable. Throughout, the integer indices $j = 1, \ldots, m'$ and $j = m'+1, \ldots, m$ will be used as subscripts to denote the equality and inequality constraints respectively, where m' and m are non-negative integers. $\check{x}_i$ and $\hat{x}_i$ denote the lower and upper bounds on a primal variable x_i respectively, ensuring that $\mathbf{x} \in \mathcal{C} \subset \mathcal{R}^n$ is bounded and closed, where $\mathcal{C}$ is the set containing the aforementioned primal bounds.

Although many techniques exist to solve $\mathcal{P}_{\text{NLP}}$ in (5.1), we shall use what is arguably considered state-of-the-art in structural optimization, namely sequential approximate optimization (SAO). SAO relies upon the iterative solution of approximate optimization problems $\mathcal{P}_{\text{P}}[k]$, with $k = 0, 1, 2, \ldots$, constructed around a primal iterate $\mathbf{x}^k$, such that

$$\begin{aligned} \min_{\mathbf{x}} \ & \tilde{f}_0^k(\mathbf{x}) \\ \text{subject to} \ & \tilde{f}_j^k(\mathbf{x}) = 0, \qquad j = 1, \ldots, m', \\ & \tilde{f}_j^k(\mathbf{x}) \leq 0, \qquad j = m'+1, \ldots, m, \\ & \check{x}_i^k \leq x_i \leq \hat{x}_i^k, \qquad i = 1, \ldots, n. \end{aligned} \tag{5.2}$$

The continuous approximate primal problem $\mathcal{P}_{\text{P}}[k]$ in (5.2) contains n primal unknown variables, m' equality constraints, $m - m'$ inequality constraints and $2n$ bound constraints. In sections to come, we demonstrate that bound constraints are handled efficiently in our algorithm, by exploiting a strategy inspired by the Falk dual [70].

A notable feature of SAO is that $\mathcal{P}_{\text{P}}[k]$ is intentionally comprised of *separable* and *convex* approximate functions $\tilde{f}_j^k(\mathbf{x})$, since second-order information is often prohibitively expensive for large-scale problems, with the Lagrangian Hessian requiring $\mathcal{O}(n^2)$ elements. Instead, so-called intervening variables are exploited to construct *diagonal* Hessian terms in a second-order Taylor series. Hopefully, the intervening variables capture sufficient second-order information of $\mathcal{P}_{\text{NLP}}$, while requiring only $\mathcal{O}(n)$ elements; assuming that the physics of the problem is understood and appropriate intervening variables are used. The intervening variables then become first-order Taylor series in $\mathcal{P}_{\text{P}}[k]$, allowing for efficient updates following the solution of $\mathcal{P}_{\text{P}}[k]$. A popular intervening variable in structural optimization is $z_i = x_i^{-1}$, which captures the inverse relationship between stress and area.

Examples of popular algorithms that exploit separable approximations include the convex linearization algorithm (CONLIN) of Fleury and Braibant [10], as well as the method of moving asymptotes (MMA) by Svanberg [8, 9]; the latter being a generalization of the

former. Herein, we are most interested in the approximations proposed by Groenwold and Etman [11], since their approximations are able to capture reciprocal-like behavior by relying upon so-called 'approximated-approximations' [38], where the quadratic approximation is constructed to the reciprocal approximation. Indeed, the 'approximated-approximations' can be constructed to many other approximations, including the aforementioned CONLIN and MMA algorithms, while retaining the desired separability of $\mathcal{P}_{\mathrm{P}}[k]$.

The subproblems $\mathcal{P}_{\mathrm{P}}[k]$ used in structural optimization are often solved in the *dual* space, which is theoretically problem-free under the assumption that the approximate functions are both *separable* and *convex*. Dual methods are especially popular in the sole presence of inequality constraints, hence many structural problems of interest are solved for using pure dual methods. Furthermore, the primal separability of $\mathcal{P}_{\mathrm{P}}[k]$ allows for closed-form primal updates (minimizers) in dual methods [70], greatly increasing the efficiency of such methods for large-scale problems, especially when $m \ll n$. However, although closed-form primal updates exist for pure dual methods, the dual subproblems traditionally require the use of iterative solvers, possibly requiring prohibitively expensive computational effort. For a popular dual statement such as the Falk dual, the dual subproblems are often not trivial to solve for, since the iterative solver must handle both bound constraints and second-order discontinuities [71]. Notwithstanding the lack of primal and dual separability in pure dual statements, *both* dual and primal separability exist when a sequential quadratic programming (SQP) formulation is used instead, as we will discuss in what is to follow.

It is well known that for convex and separable functions, closed-form minimizers exist for both the primal and dual variables [70, 72, 73, 74, 75], etc. However, to the best of our knowledge, we are unaware of applications in structural optimization that exploit both closed-form primal and dual variable updates, possibly since bound (box) and inequality constraints introduce difficulties in implementing these methods. Hence, iterative solvers are typically used to solve for the primal and/or dual variables, since provision can be made to accommodate bound and inequality constraints, provided a suitable solver is used. Furthermore, we are also unaware of SQP-like methods that exploit both intervening variables *and* closed-form expressions for the primal and dual variables. The only SQP-like method that satisfies the aforementioned properties is our work in [35], namely a separable augmented Lagrangian algorithm (SALA) for optimal structural design, derived from the class of alternating directions of multiplier method (ADMM) type algorithms. SALA is able to exploit intervening variables *and* has the salient feature of embarrassingly parallel, closed-form primal and dual variable updates, although the updates need to be performed numerous times per subproblem. Our separable Lagrangian algorithm (SLA) proposed herein is fundamentally different to SALA, since it does not require an augmented Lagrangian formulation, thereby circumventing the difficulties associated with choosing an appropriate penalty parameter. In SALA, the penalty parameter creates many difficulties, since it acts as a scaling parameter on the subproblem level [49, 46]. Lastly, while ADMM methods are usually only suitable for modest levels of accuracy [35, 46], `SLA` is able to achieve high levels of accuracy comparable to state-of-the-art solvers; a result we demonstrate in sections to come.

Both the primal and dual variables are separable in `SLA`, hence `SLA` is ideal for implementation on massively parallel computational devices, such as general purpose graphical compute units (GPGPUs). The primal updates are embarrassingly parallel and can be performed as a single

matrix-vector operation, ideal for modern GPGPUs. The dual updates are better solved as the solution to a positive-definite linear system, which depending on the sparsity, may benefit to varying degrees from a GPGPU implementation. Naturally, solving the dual linear system may require greater computational complexity compared to updating the primal variables; since solving the dual linear system requires at most $\mathcal{O}(m^3)$ operations[1], whereas updating the primal variables requires $\mathcal{O}(mn)$ operations.

To demonstrate the efficiency of `SLA`, we provide numerical results for large-scale structural problems involving hundreds of millions of variables and constraints, which neither the state-of-the-art Galahad `LSQP` [51] solver nor Falk dual can solve in reasonable time. `SLA` is able to solve these large-scale problems within a few minutes, clearly demonstrating the efficiency gained by exploiting closed-form primal and dual variable updates.

Our paper makes the following contributions: We propose a separable Lagrangian algorithm (SLA) for optimal structural design, which contains separability in both the primal and dual variables, hence `SLA` is free from any form of search method. Furthermore, `SLA` is able to exploit intervening variables, allowing for the efficient solution of large-scale structural optimization problems. Although we do not demonstrate it herein, the primal-dual separability makes `SLA` ideal for implementation on massively parallel computational devices, such as GPGPUs.

Our paper is arranged as follows: In Section 5.3, we introduce the separable and convex approximate functions, along with the direct and intervening variables that are used throughout. The dual of Falk and quadratic-like dual subproblems are presented in Section 5.4, followed by the development of `SLA` in Section 5.5. In Section 5.6, we introduce an efficient primal-dual active set strategy for inequality constraints, before providing brief comments regarding global convergence of `SLA`. We then present numerical results in Section 5.7 and conclusions in Section 5.8, while offering some thoughts for future work.

5.3 Quadratic-like approximations

To construct the separable and convex approximate functions, we make use of an incomplete series expansion (ISE) [2]. Given an iteration point $\mathbf{x}^k$, arising from the solution of iterate $k-1$, the approximate objective and constraint functions $\tilde{f}_j^k(\mathbf{x})$ are constructed, such that

$$\tilde{f}_j^k(\mathbf{x}) = f_j(\mathbf{x}^k) + \left(\frac{\partial f_j(\mathbf{x}^k)}{\partial \mathbf{x}}\right)^{\mathrm{T}} \mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}\mathbf{C}_j^k\mathbf{s}, \quad j = 0, \ldots, m. \tag{5.3}$$

In (5.3), $\mathbf{s} = (\mathbf{x} - \mathbf{x}^k)$ and $\mathbf{C}_j^k$ is an approximate *diagonal* Hessian matrix, containing approximate second-order information for an approximate function $\tilde{f}_j^k(\mathbf{x})$. For notational simplicity

[1] We note that $\mathcal{O}(m^3)$ operations is a conservative estimate, since the dual linear system requires the solution to a positive-definite matrix.

we will abbreviate approximate information, as

$$f_j^k = f_j(\mathbf{x}^k),$$
$$\left(\frac{\partial f_j}{\partial x_i}\right)^k = \frac{\partial f_j(\mathbf{x}^k)}{\partial x_i},$$

allowing us to simplify the notation for the approximate functions, whereby

$$\tilde{f}_j^k(\mathbf{x}) = f_j^k + \sum_{i=1}^{n} \left(\frac{\partial f_j}{\partial x_i}\right)^k (x_i - x_i^k) + \frac{1}{2}\sum_{i=1}^{n} c_{2i_j}^k (x_i - x_i^k)^2. \tag{5.4}$$

The c_{2i_j} terms, comprised of the direct or intervening variables, are used to construct the diagonal elements of $\mathbf{C}_j^k$. We consider only two simple instances of second-order approximations herein, in which c_{2i_j} is chosen as:

1. A spherical quadratic approximation proposed by Snyman and Hay [54], denoted by SPH-QDR, such that the approximate and real function values are equal at the previous iterate, i.e. $f_j^{k-1} = \tilde{f}_j^{k-1}$.
2. The quadratic approximation to the reciprocal approximation presented in [38], which closely resembles the popular MMA [8] approximations, denoted by T2:R.

Although many other approximations exist, see [38] for examples, we are only concerned with the above mentioned SPH-QDR and T2:R approximations herein.

Transforming $\mathcal{P}_{\text{NLP}}$ into a sequential quadratic program (SQP) $\mathcal{P}_{\text{PQ}}[k]$ is trivial, since the approximations (5.3) are already cast in quadratic form. Hence, we may express $\mathcal{P}_{\text{PQ}}[k]$ as

□ *Quadratic approximate program* $\mathcal{P}_{\text{PQ}}[k]$

$$\begin{aligned}
\min_{\mathbf{s}} \ & \tilde{f}_0^k(\mathbf{s}) = f_0^k + \boldsymbol{\nabla} f_0^{k\mathrm{T}}\mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}\mathbf{Q}^k\mathbf{s} \\
\text{subject to } & \tilde{f}_j^k(\mathbf{s}) = f_j^k + \boldsymbol{\nabla} f_j^{k\mathrm{T}}\mathbf{s} = 0, && j = 1,\ldots,m', \\
& \tilde{f}_j^k(\mathbf{s}) = f_j^k + \boldsymbol{\nabla} f_j^{k\mathrm{T}}\mathbf{s} \leq 0, && j = m'+1,\ldots,m, \\
& \check{x}_i^k \leq x_i \leq \hat{x}_i^k, && i = 1,\ldots,n,
\end{aligned} \tag{5.5}$$

where

$$\boldsymbol{\nabla} f_j^k = \left[\left(\frac{\partial f_j}{\partial x_1}\right)^k, \ldots, \left(\frac{\partial f_j}{\partial x_n}\right)^k\right]^{\mathrm{T}}, \qquad j = 0,\ldots,m,$$

with $\mathbf{Q}^k$ the Hessian matrix of the approximate Lagrangian $\tilde{\mathcal{L}}^k$, formed from the approximations in (5.3). The interested reader is referred to Etman *et al.* [52, 53] for more details.

To construct $\mathbf{Q}^k$, we use the Lagrange multipliers, or dual variables $\boldsymbol{\mu}^k$, from the previous iteration point, since the dual variables at the solution point $\boldsymbol{\mu}^{k+1}$ of $\mathcal{P}_{\text{PQ}}[k]$ are unknown in advance. $\mathbf{Q}^k$ is then constructed by letting

$$B_{ii}^k = c_{2i_0}^k + \sum_{j=1}^{m} \mu_j^k c_{2i_j}^k, \quad i = 1,\ldots,n, \tag{5.6}$$

with $B_{id}^k = 0 \; \forall \; i \neq d$, i and $d = 1, \ldots, n$. From hereon, μ_j for $j = 1, \ldots, m'$ and $j = m' + 1, \ldots, m$ represents the dual variables associated with the equality and inequality constraints respectively.

To ensure that a unique solution exists for (5.5), $\mathbf{Q}^k$ must be (semi) positive-definite. For a diagonal matrix $\mathbf{Q}^k$, enforcing positive-definiteness requires that each diagonal element Q_{ii}^k be positive. Since μ_j^k and c_{2i_j} in (5.6) are unrestricted in sign, it is possible that $B_{ii}^k < 0$; hence, we enforce

$$Q_{ii}^k = \begin{cases} B_{ii}^k & \text{if} \quad B_{ii}^k > \epsilon \\ \epsilon & \text{if} \quad B_{ii}^k \leq \epsilon \end{cases}, \quad i = 1, \ldots, n, \tag{5.7}$$

with $\epsilon > 0$ prescribed and of 'small' magnitude.

5.4 A quadratic-like dual subproblem

We depart with the well-known Lagrangian function for the approximate quadratic program $\mathcal{P}_{\text{PQ}}[k]$ in (5.5), given by

$$\tilde{\mathcal{L}}^k(\mathbf{x}, \boldsymbol{\mu}) = \tilde{f}_0^k(\mathbf{x}) + \sum_{j=1}^{m} \tilde{f}_j^k(\mathbf{x}) \mu_j. \tag{5.8}$$

Since the approximate functions in $\mathcal{P}_{\text{PQ}}[k]$ are convex and separable, the dual of Falk [70] is an ideal candidate to solve for (5.8). The dual function $\tilde{\gamma}$, which Falk refers to as the auxiliary function, is defined by Falk [70] as

$$\begin{aligned} \tilde{\gamma}(\boldsymbol{\mu}) = \quad & \min_{\mathbf{x}} \tilde{\mathcal{L}}^k(\mathbf{x}, \boldsymbol{\mu}) \\ \text{subject to} \quad & \mathbf{x} \in \mathcal{C}, \end{aligned} \tag{5.9}$$

where the set $\mathcal{C}$ is bound and closed, consisting of the primal lower and upper bounds $\check{x}_i$ and $\hat{x}_i$ respectively. The domain of $\tilde{\gamma}$, $\mathcal{D}[\tilde{\gamma}]$, is then defined over all $\boldsymbol{\mu}$ where $\tilde{\mathcal{L}}^k(\mathbf{x}, \boldsymbol{\mu})$, $\mathbf{x} \in \mathcal{C}$ attains a finite minimum with respect to $\mathbf{x}$, hence the primal minimizer $\mathbf{x}(\boldsymbol{\mu})$ is a function of $\boldsymbol{\mu}$. Despite the minimum $\mathbf{x}(\boldsymbol{\mu})$ not necessarily being unique, the use of separable and convex approximate functions ensures a unique primal-dual minimizer pair in (5.9). Thus, it follows that the dual problem is given by

$$\begin{aligned} \max_{\boldsymbol{\mu}} \; & \tilde{\gamma}(\boldsymbol{\mu}) \\ \text{subject to} \quad & \mu_j \geq 0, \qquad j = m' + 1, \ldots, m, \end{aligned} \tag{5.10}$$

where the dual variables associated with the equality constraints, μ_j, $j = 1, \ldots, m'$, may take any sign. Since the primal minimizer of (5.9) is $\mathbf{x}(\boldsymbol{\mu})$, it is convenient to rewrite the dual function $\tilde{\gamma}$ (5.9), as

$$\tilde{\gamma}(\boldsymbol{\mu}) = \tilde{f}_0^k(\mathbf{x}(\boldsymbol{\mu})) + \sum_{j=1}^{m} \tilde{f}_j^k(\mathbf{x}(\boldsymbol{\mu})) \mu_j, \tag{5.11}$$

allowing for the dual problem to be solely expressed in terms of $\boldsymbol{\mu}$, such that

$$\begin{aligned} \max_{\boldsymbol{\mu}} \ & \tilde{\gamma}(\boldsymbol{\mu}) = \tilde{f}_0^k(\mathbf{x}(\boldsymbol{\mu})) + \sum_{j=1}^{m} \tilde{f}_j^k(\mathbf{x}(\boldsymbol{\mu}))\mu_j \\ \text{subject to} \quad & \mu_j \geq 0, \qquad j = m' + 1, \ldots, m, \\ & \mathbf{x}(\boldsymbol{\mu}) \in \mathcal{C}. \end{aligned} \tag{5.12}$$

To obtain an expression for $\mathbf{x}(\boldsymbol{\mu})$, the stationary conditions of (5.9) are taken with respect to $\mathbf{x}$, which leads to Falk's efficient closed-form primal update [70] in QP form [71], given by

$$x_i(\boldsymbol{\mu}) = \begin{cases} \beta_i(\boldsymbol{\mu}) & \text{if} \quad \check{x}_i^k < \beta_i(\boldsymbol{\mu}) < \hat{x}_i^k, \\ \hat{x}_i^k & \text{if} \quad \beta_i(\boldsymbol{\mu}) \geq \hat{x}_i^k, \\ \check{x}_i^k & \text{if} \quad \beta_i(\boldsymbol{\mu}) \leq \check{x}_i^k, \end{cases}$$

where

$$\beta_i(\boldsymbol{\mu}) = x_i^k - \left(c_{2i_0}^k + \sum_{j=1}^{m} \mu_j^k c_{2i_j}^k \right)^{-1} \left(\left(\frac{\partial f_0}{\partial x_i} \right)^k + \sum_{j=1}^{m} \left(\frac{\partial f_j}{\partial x_i} \right)^k \mu_j \right).$$

For the sake of brevity in what is to follow, we rewrite the primal minimizer as

$$x_i(\boldsymbol{\mu}) = \Pi \left(x_i^k - \left(c_{2i_0}^k + \sum_{j=1}^{m} \mu_j^k c_{2i_j}^k \right)^{-1} \left(\left(\frac{\partial f_0}{\partial x_i} \right)^k + \sum_{j=1}^{m} \left(\frac{\partial f_j}{\partial x_i} \right)^k \mu_j \right) \right), \qquad i = 1, \ldots, n, \tag{5.13}$$

where the operator $\Pi(\cdot)$ is the projection onto $[\check{x}_i, \hat{x}_i]$, thereby ensuring the primal bounds are respected, i.e. $\mathbf{x} \in \mathcal{C}$. Note that (5.13) is linear with respect to $\boldsymbol{\mu}$, since the μ_j^k arise from the previous iterate k and are constant. In a pure dual statement, which is commonly employed for the Falk dual, the μ_j^k in the denominator of (5.13) is replaced by μ_j, hence (5.13) is not linear with respect to $\boldsymbol{\mu}$ for pure dual statements. The interested reader is referred to [71] and [74] for more details.

The Falk dual problem in (5.12) requires the *iterative* solution of the m dual variables $\boldsymbol{\mu} \in \mathcal{R}^m$ [71], typically through the use of a solver that efficiently handles bound constraints and second-order discontinuities, such as `l-BFGS-b` [4]. Incorporating the primal bounds into the dual problem results in a 'piecewise quadratic' dual topology [71, 76], which may be computationally expensive to maximize, as noted by Fleury [76].

For problems where $m \ll n$, solving (5.12) in the so-called 'dual space' is extremely efficient for pure dual statements [52], especially since the primal minimizer is available as a closed-form expression[2]. However, for problems where $m \approx n$ is large, solving for the dual variables $\boldsymbol{\mu} \in \mathcal{R}^m$ using pure dual methods may be prohibitively expensive. This result was shown by Etman *et al.* in [52], where the authors additionally demonstrated that QP-like methods are more suitable for large-scale problems involving $n \approx m$ variables and constraints.

In this study, we demonstrate that solving a *continuous* QP variant of the Falk dual is efficient for large-scale problems. A continuous dual problem may be formed by solving an

[2]We make use of the bounded dual of Wood *et al.* [6] for the Falk dual implementation used herein.

unconstrained variant of the dual problem in (5.12), which results in closed-form expressions for the primal and dual variable updates. Although the unconstrained dual problem may require multiple solutions if the primal and/or dual bounds are violated, as opposed to the single solution of the piecewise quadratic pure dual problem; each solution requires minimal computational effort as a result of closed-form expressions being available for the primal and dual variable updates.

5.5 A quadratic-like separable Lagrangian algorithm

To demonstrate that a closed-form solution exists for *both* the dual and primal variables, we require an unconstrained formulation of the dual problem in (5.12). Consider the requirement that the primal variables reside in the closed and bound space $\mathcal{C}$, given by $\mathbf{x}(\boldsymbol{\mu}) \in \mathcal{C}$, which requires the projection $\Pi(\mathbf{x}(\boldsymbol{\mu}))$. If the solution $\mathbf{x}^*$ is an *interior-point* of $\mathcal{C}$, the projection $\Pi(\mathbf{x}(\boldsymbol{\mu}^*))$ will not be required at the solution, resulting in $\mathbf{x}(\boldsymbol{\mu}^*) = \Pi(\mathbf{x}(\boldsymbol{\mu}^*))$. Indeed, this can trivially be extended to a subproblem. If $\mathbf{x}(\boldsymbol{\mu}^{k*}) = \Pi(\mathbf{x}(\boldsymbol{\mu}^{k*}))$, the projection need not be enforced for a given subproblem k at the solution $k*$. For computational efficiency, we first solve for $\boldsymbol{\mu}^{k*}$, before evaluating whether the projection $\Pi(\mathbf{x}(\boldsymbol{\mu}^{k*}))$ is required, allowing us to remove the constraint $\mathbf{x}(\boldsymbol{\mu}) \in \mathcal{C}$ in (5.12). Therefore, this section is divided into two cases; namely if the projection $\Pi(\mathbf{x}(\boldsymbol{\mu}^{k*}))$ is *not* required, followed by if the projection $\Pi(\mathbf{x}(\boldsymbol{\mu}^{k*}))$ is required.

5.5.1 Primal projection free subproblems

After assuming that the primal projection $\Pi(\cdot)$ is not required, we address the constraint that the Lagrange multipliers associated with the inequality constraints must reside in $(\mathcal{R}^m)^+$, such that $\mu_j \geq 0$ for $j = m' + 1, \ldots, m$. Hence, we use an active-set strategy to ensure $\mu_j \geq 0,\ j \in \mathcal{J}$ holds for constraint indices j in the active-set $\mathcal{J}$. For constraints that are not included in the active-set, we enforce $\mu_j = 0,\ j \notin \mathcal{J}$; ensuring complementary slackness in the well-known KKT conditions.

After relaxing the constraints on Falk's dual problem (5.12), we proceed with an unconstrained formulation of the dual problem, given by

$$\max_{\boldsymbol{\mu}}\ \tilde{\phi}(\boldsymbol{\mu}) = \tilde{f}_0^k(\mathbf{x}(\boldsymbol{\mu})) + \sum_{j \in \mathcal{J}} \tilde{f}_j^k(\mathbf{x}(\boldsymbol{\mu}))\mu_j. \tag{5.14}$$

For notational simplicity and brevity in what is to follow, we introduce matrix and vector notation for the approximate and original functions, as opposed to using summation convention. Let $\mathcal{J}$ denote the active-set, the composition of which is discussed in Section 5.6, where $\mathcal{J}(1), \ldots, \mathcal{J}(m_a)$ are the monotonically increasing integer indices that map to the $m_a \leq m$ active constraints. For example, if $j = 5$ is the first active constraint in $\mathcal{J}$, then $\mathcal{J}(1) = 5$, or if $j = 5$ is the last active constraint in $\mathcal{J}$, then $\mathcal{J}(m_a) = 5$. This notation will interchangeably be used between approximate and original functions throughout this section, with the intended meaning clear from the context. Let the vector of active approximate constraints

be represented by

$$\tilde{f}_{\mathcal{J}}^{k}(\mathbf{x}) = \left[\tilde{f}_{\mathcal{J}(1)}^{k}(\mathbf{x}), \ldots, \tilde{f}_{\mathcal{J}(m_a)}^{k}(\mathbf{x})\right]^{\mathrm{T}} \quad \in \mathcal{R}^{m_a \times 1},$$

which can be once continuously differentiated to form the active-set constraint Jacobian matrix, as

$$\boldsymbol{\nabla} \tilde{f}_{\mathcal{J}}^{k} = \begin{bmatrix} — & \boldsymbol{\nabla} \tilde{f}_{\mathcal{J}(1)}^{k} & — \\ — & \vdots & — \\ — & \boldsymbol{\nabla} \tilde{f}_{\mathcal{J}(m_a)}^{k} & — \end{bmatrix}, \quad \in \mathcal{R}^{m_a \times n}.$$

We require that the rows $\boldsymbol{\nabla} \tilde{f}_{\mathcal{J}(1)}^{k}, \ldots, \boldsymbol{\nabla} \tilde{f}_{\mathcal{J}(m_a)}^{k}$ are linearly independent, thereby satisfying the KKT conditions that are implied throughout. The Lagrange multipliers associated with the active constraints, $\boldsymbol{\mu}_{\mathcal{J}}$, are used to construct the Hessian from the active-set, such that

$$Q_{ii,\mathcal{J}}^{k} = \begin{cases} B_{ii,\mathcal{J}}^{k} & \text{if} \quad B_{ii,\mathcal{J}}^{k} > \epsilon \\ \epsilon & \text{if} \quad B_{ii,\mathcal{J}}^{k} \leq \epsilon \end{cases}, \quad i = 1, \ldots, n,$$

where

$$B_{ii,\mathcal{J}}^{k} = c_{2i_0}^{k} + \sum_{j \in \mathcal{J}} \mu_j^{k} c_{2i_j}^{k}, \quad i = 1, \ldots, n,$$

with the closed-form primal minimizer given in the new notation, as

$$\mathbf{x}(\boldsymbol{\mu}_{\mathcal{J}}) = \mathbf{x}^{k} - \mathbf{E}_{\mathcal{J}}^{k} \left(\boldsymbol{\nabla} f_0^{k} + \boldsymbol{\nabla} f_{\mathcal{J}}^{k\,\mathrm{T}} \boldsymbol{\mu}_{\mathcal{J}} \right), \quad \in \mathcal{R}^{n}.$$

Here, $\mathbf{E}_{\mathcal{J}}^{k} = \left(\mathbf{Q}_{\mathcal{J}}^{k}\right)^{-1}$ is used for brevity and clarity. Finally, the unconstrained dual problem in (5.14) can be written in matrix notation, as

$$\max_{\boldsymbol{\mu}_{\mathcal{J}}} \; \tilde{\phi}(\boldsymbol{\mu}_{\mathcal{J}}) = \tilde{f}_0^{k}(\mathbf{x}(\boldsymbol{\mu}_{\mathcal{J}})) + \tilde{f}_{\mathcal{J}}^{k}(\mathbf{x}(\boldsymbol{\mu}_{\mathcal{J}}))^{\mathrm{T}} \boldsymbol{\mu}_{\mathcal{J}}. \tag{5.15}$$

After some matrix calculus, once continuously differentiating the concave function (5.15) with respect to $\boldsymbol{\mu}_{\mathcal{J}}$, yields

$$\left(\boldsymbol{\nabla} f_{\mathcal{J}}^{k} \mathbf{E}_{\mathcal{J}}^{k} \boldsymbol{\nabla} f_{\mathcal{J}}^{k\,\mathrm{T}} \right) \boldsymbol{\mu}_{\mathcal{J}}^{*} = f_{\mathcal{J}}^{k} - \boldsymbol{\nabla} f_{\mathcal{J}}^{k} \mathbf{E}_{\mathcal{J}}^{k} \boldsymbol{\nabla} f_0^{k}, \tag{5.16}$$

allowing for the efficient closed-form dual variable solution $\boldsymbol{\mu}_{\mathcal{J}}^{*}$. Notice that all the approximate functions in (5.16) have been dropped, except the approximate Hessian term, with the original functions used instead. The first-order conditions of the dual problem, given by (5.16), may alternatively be obtained by evaluating the constraint functions at $\tilde{f}_{\mathcal{J}}^{k}(\mathbf{x}(\boldsymbol{\mu}^{*})) = 0$. However, for the sake of clarity, we prefer to derive said conditions from first principles.

The linear system in (5.16) is positive-definite, under the assumption an appropriate active-set strategy is used. This can be can be verified by noting that

$$\mathbf{v}^{\mathrm{T}} \left(\boldsymbol{\nabla} f_{\mathcal{J}}^{k} \mathbf{E}_{\mathcal{J}}^{k} \boldsymbol{\nabla} f_{\mathcal{J}}^{k\,\mathrm{T}} \right) \mathbf{v} > 0 \quad \forall \quad \mathbf{v} \in \mathcal{R}^{m_a} \setminus \mathbf{0},$$

thus guaranteeing a unique solution for $\boldsymbol{\mu}^*_{\mathcal{J}}$ exists, and allowing for efficient exploitation by linear solvers. Solving (5.16) as a linear system is preferred to finding the inverse, which may be prohibitively expensive in computational complexity and memory requirements.

Following the update in (5.16), bounds are placed on the dual variables. This computationally efficient technique was successfully implemented by Wood *et al.* [6] for the Falk dual, acting as a form of relaxation for infeasible dual subproblems. Indeed, bounding the dual variables herein is a simple projection, given by

$$\begin{aligned} \mu^l_j &= \Lambda(\mu^*_j), \qquad && j \in \mathcal{J}, \\ \mu^l_j &= 0, && j \notin \mathcal{J}, \end{aligned} \tag{5.17}$$

where $\Lambda(\cdot)$ is the projection onto $[\check{\mu}_j, \hat{\mu}_j]$. The superscript l is used to denote a subproblem iteration, many of which may be required if the primal projection is enforced. Hence, we introduce the subproblem iteration notation in this section, since the following section will make extensive use of it. The upper and lower bounds for the dual variables are chosen as

$$\check{\mu}_j = \begin{cases} -\hat{\mu}_j, & j = 1, \ldots, m', \\ 0, & j = m'+1, \ldots, m, \end{cases}$$

with $\hat{\mu}_j = 10^{12}$ for $j = 1, \ldots, m$ chosen for our numerical testing. Finally, after the closed-form dual update has been performed, the closed-form primal update follows, as

$$\mathbf{x}^l = \mathbf{x}^k - \mathbf{E}^k_{\mathcal{J}} \left(\boldsymbol{\nabla} f^k_0 + \boldsymbol{\nabla} f^{k\,\mathrm{T}}_{\mathcal{J}} \boldsymbol{\mu}^l_{\mathcal{J}} \right). \tag{5.18}$$

5.5.2 Enforcing subproblem primal projections

To determine if the primal projection $\Pi(\cdot)$ is required, we test if $\mathbf{x}^l = \Pi(\mathbf{x}^l)$. Indeed, if the aforementioned equality is satisfied, the primal projection is not required and $(\mathbf{x}^l, \boldsymbol{\mu}^l) = (\mathbf{x}^{k+1}, \boldsymbol{\mu}^{k+1})$ is the primal-dual subproblem solution. However, should $\mathbf{x}^l \neq \Pi(\mathbf{x}^l)$, the primal variables that are modified by the projection $\Pi(\cdot)$ are removed from the subproblem. The removed primal variables are included in a modified QP subproblem as projected constant terms, in a similar fashion to the Falk dual; where primal variables requiring the projection $\Pi(\cdot)$ attain constant values, since $x_i(\boldsymbol{\mu}^l_{\mathcal{J}}) \notin \mathcal{C}$ will result in a projection onto either $\check{x}_i$ or $\hat{x}_i$. Hence, if a projection is required for a primal variable x^l_i, we enforce the projection $\Pi(x^l_i)$, add the variable as a constant term to a modified QP problem and resolve the modified QP problem. This process is repeated until all primal solutions of some modified QP problem are interior-points of $\mathcal{C}$, none of which require the projection $\Pi(\cdot)$.

Although n modified QP problems may arise in theory if only a single variable x^l_i is held constant per iteration l, our numerical results have demonstrated that few modified QP problems are constructed per subproblem, even if many of the primal variables reside on either $\check{x}_i$ or $\hat{x}_i$ at the solution[3]. Instead, solving for the primal-dual solutions in the proposed unconstrained, closed-form fashion results in minimal subproblem effort, even for large-scale

[3]We demonstrate this by means of a topology optimization problem in Section 5.7, whereby many of the primal variables reside on the upper or lower bound at the solution.

problems where $n \approx m$. Furthermore, although our chosen projection strategy may not be 'optimal' for a given subproblem, our numerical results demonstrate a competitive number of iterations k compared to state-of-the-art solvers, which solve the subproblems in a more rigorous manner.

To determine the set of primal variables that do not require the projection $\Pi(\cdot)$, let

$$\vec{\mathbf{x}} = x_i \; \forall \; i \in \mathcal{I}^l, \tag{5.19}$$

where

$$\mathcal{I}^l = \Big\{ i : \quad \check{x}_i^k < x_i^l < \hat{x}_i^k. \tag{5.20}$$

The primal variables $\vec{\mathbf{x}}$ will be referred to as the 'free' variables, with the variables $x_i, \; i \notin \mathcal{I}^l$ denoted as the 'fixed' variables. For the fixed variables, we apply the projection

$$x_i^{k+1} = \Pi(x_i^l), \quad i \notin \mathcal{I}^l, \tag{5.21}$$

and remove them from a subproblem k, hence fixed variables cannot become free variables for a given iteration k. All primal variables are free variables at the start of a given subproblem k. To determine the number of variables unmodified by $\Pi(\cdot)$, let

$$\zeta^l = \text{card}\left(\mathcal{I}^l\right), \tag{5.22}$$

denote the cardinality, or number of free primal variables, in the set $\mathcal{I}^l$.

The free and fixed variables allow us to reconstruct the approximate objective and constraint functions, forming a modified QP problem, given by

$$\begin{aligned}
\min_{\vec{\mathbf{x}}} \; & \vec{f}_0^k(\vec{\mathbf{x}}) = f_0^k + \sum_{i \in \mathcal{I}^l} \left(\frac{\partial f_0}{\partial x_i}\right)^k (x_i - x_i^k) + \frac{1}{2} \sum_{i \in \mathcal{I}^l} Q_{ii}^k (x_i - x_i^k)^2 + \Phi^l \\
\text{subject to } & \vec{f}_j^k(\vec{\mathbf{x}}) = f_j^k + \sum_{i \in \mathcal{I}^l} \left(\frac{\partial f_j}{\partial x_i}\right)^k (x_i - x_i^k) + \Psi_j^l = 0, \qquad j = 1, \ldots, m', \\
& \vec{f}_j^k(\vec{\mathbf{x}}) = f_j^k + \sum_{i \in \mathcal{I}^l} \left(\frac{\partial f_j}{\partial x_i}\right)^k (x_i - x_i^k) + \Psi_j^l \leq 0, \qquad j = m'+1, \ldots, m, \\
& \check{x}_i \leq x_i \leq \hat{x}_i, \qquad i \in \mathcal{I}^l,
\end{aligned} \tag{5.23}$$

where

$$\Phi^l = \sum_{i \notin \mathcal{I}^l} \left(\frac{\partial f_0}{\partial x_i}\right)^k (x_i^{k+1} - x_i^k) + \frac{1}{2} \sum_{i \notin \mathcal{I}^l} Q_{ii}^k (x_i^{k+1} - x_i^k)^2, \tag{5.24}$$

$$\Psi_j^l = \sum_{i \notin \mathcal{I}^l} \left(\frac{\partial f_j}{\partial x_i}\right)^k (x_i^{k+1} - x_i^k). \tag{5.25}$$

The modified QP problem in (5.23) is repeatedly constructed and solved for $l = 1, \ldots,$ until the primal projection $\Pi(\cdot)$ is not required for any of the free variables (i.e. $\zeta^l = \zeta^{l-1}$),

resulting in all the variables $x_i,\ \forall\ i \in \mathcal{I}^l$ being interior-point solutions of $\mathcal{C}$. The dual and primal updates of (5.23) are performed using Equations (5.16) and (5.18) respectively, using the free variables and the modified dual problem formed from the modified QP problem in (5.23). Since (5.16) and (5.18) are constructed from separable functions, it is trivial to modify the updates to solve for (5.23). The dual update for the modified QP problem is given by

$$\left(\boldsymbol{\nabla}\vec{f}^k_{\mathcal{J}}\vec{\mathbf{E}}^k_{\mathcal{J}}\boldsymbol{\nabla}\vec{f}^{k\mathrm{T}}_{\mathcal{J}}\right)\boldsymbol{\mu}^*_{\mathcal{J}} = f^k_{\mathcal{J}} + \Psi^l_{\mathcal{J}} - \left(\boldsymbol{\nabla}\vec{f}^k_{\mathcal{J}}\vec{\mathbf{E}}^k_{\mathcal{J}}\boldsymbol{\nabla}\vec{f}^k_0\right), \tag{5.26}$$

where $\boldsymbol{\nabla}\vec{f}^k_{\mathcal{J}}$ and $\vec{\mathbf{E}}^k_{\mathcal{J}}$ indicates the respective Jacobian and Hessian matrices formed from the separable terms $x_i, i \in \mathcal{I}^l$ and the active-set $\mathcal{J}$, at an iteration k. The same notation is used for both $\boldsymbol{\nabla}\vec{f}^k_0$ and $\vec{\mathbf{x}}^k$, albeit that the active-set strategy is not applied to these terms. After determining $\boldsymbol{\mu}^l_{\mathcal{J}} = \Lambda(\boldsymbol{\mu}^*_{\mathcal{J}})$, the free primal variables are updated, as

$$\vec{\mathbf{x}}^l = \vec{\mathbf{x}}^k - \vec{\mathbf{E}}^k_{\mathcal{J}}\left(\boldsymbol{\nabla}\vec{f}^k_0 + \boldsymbol{\nabla}\vec{f}^{k\mathrm{T}}_{\mathcal{J}}\boldsymbol{\mu}^l_{\mathcal{J}}\right). \tag{5.27}$$

An important note for the implementation of a move limit: The modified QP problem in (5.23) only needs to be constructed if the move limit upper or lower bound, $\hat{x}^k_i$ or $\check{x}^k_i$, coincides with the true upper and lower bounds $\hat{x}_i$ and $\check{x}_i$. Since our proposed method converges to interior-points of $\mathcal{C}$, which have defined KKT points, invoking the modified QP problem in (5.23) is only required if the subproblem solution resides on the boundary of $\mathcal{C}$, where the KKT conditions are undefined. Hence, we only require the use of (5.23) if $\Pi(\cdot)$ is required *and* the move limit upper or lower bounds will result in a projection onto either $\hat{x}_i$ or $\check{x}_i$. For large-scale problems that use move limits, less subproblem evaluations l are required using this strategy, with minimal impact on iteration count k. Indeed, this results in less total subproblem evaluations, therefore minimizing the time-complexity required for the solution.

Since we solve for the solution of the subproblem KKT system, a Newton step [5] is taken for the primal (5.27) and dual (5.26) variable updates. Furthermore, the primal-dual active-set strategy used results in a semi-smooth Newton method, described in detail in [77]. However, unlike other methods that exploit a Newton step, our proposed method takes the primal and dual steps *independently*, greatly increasing the efficacy of the method for large-scale problems.

In terms of the computational efficiency, the primal updates (5.27) can be performed in an embarrassingly parallel fashion, requiring $\mathcal{O}(nm_a)$ operations[4] for a single matrix-vector product. Forming the linear system for the dual updates in (5.26) requires $\mathcal{O}(n)$ operations to invert the diagonal Hessian $\mathbf{Q}^k_{\mathcal{J}}$, $\mathcal{O}(nm_a)$ operations to form $\boldsymbol{\nabla}f^k_{\mathcal{J}}\mathbf{E}^k_{\mathcal{J}}$, and finally a symmetric matrix-matrix multiplication of $\mathcal{O}(nm_a^2)$ operations to form $\boldsymbol{\nabla}f^k_{\mathcal{J}}\mathbf{E}^k_{\mathcal{J}}\boldsymbol{\nabla}f_{\mathcal{J}}{}^{k\mathrm{T}}$. Indeed, our proposed method is ideally suited to problems where $m \ll n$, but we demonstrate in sections to come that said method is still efficient for large-scale problems where $m \approx n$. Furthermore, all the operations are ideally suited for implementation on massively parallel computational devices, such as GPGPUs; which should see greatly diminished subproblem time-complexity for large-scale problems. We outline our method, `SLA`, in Algorithm 2.

[4]We assume complexity for operations on dense matrices herein, but we note and demonstrate that operations on sparse matrices may require significantly lower complexity.

Algorithm 2: Algorithm SLA for a *given sub*problem k

Given: $\boldsymbol{\mu}^k \in \mathcal{R}^m$, $\mathbf{x}^k \in \mathcal{R}^n$, $\boldsymbol{\nabla} f_j^k \in \mathcal{R}^n,\ j = 0, \ldots, m$, $\zeta^0 = n$, $l = 0$, $\mathcal{I}^1 = \{i : i = 1, \ldots, n\}$
while $\zeta^l \neq \zeta^{l-1}$ **do**
  $\mathcal{J} \leftarrow \mathcal{J}^l$ using (5.28)
  Construct $\boldsymbol{\nabla} \vec{f}_{\mathcal{J}}^k$ and $\vec{\mathbf{E}}_{\mathcal{J}}^k$
  Update $\boldsymbol{\mu}_{\mathcal{J}}^*$ using (5.26)
  if $\left(\mathcal{J}^{l*} \neq \mathcal{J}^l\right)$ **then**
    $\mathcal{J} \leftarrow \mathcal{J}^{l*}$ with (5.30)
    Reconstruct $\boldsymbol{\nabla} \vec{f}_{\mathcal{J}}^k$ and $\vec{\mathbf{E}}_{\mathcal{J}}^k$
    Update $\boldsymbol{\mu}_{\mathcal{J}}^*$ using (5.26)
    $\boldsymbol{\mu}^l \leftarrow \Lambda(\boldsymbol{\mu}^*)$, using (5.17)
  else
    $\boldsymbol{\mu}^l \leftarrow \Lambda(\boldsymbol{\mu}^*)$, using (5.17)
  end
  $\mathcal{I}^l$ constructed using (5.20)
  $\zeta^l \leftarrow \text{card}\left(\mathcal{I}^l\right)$ with (5.22)
  $x_i^l \leftarrow x_i(\boldsymbol{\mu}_{\mathcal{J}}^l),\ i \in \mathcal{I}^l$ using (5.27). The update for the free variables.
  $x_i^{k+1} \leftarrow \Pi(x_i(\boldsymbol{\mu}_{\mathcal{J}}^l)),\ i \notin \mathcal{I}^l$ using (5.21). The fixed variables are removed from the subproblem.
  $l \leftarrow l + 1$
end
$x_i^{k+1} \leftarrow x_i(\boldsymbol{\mu}_{\mathcal{J}}^l),\ i \in \mathcal{I}^l$ using (5.27)
$\boldsymbol{\mu}^{k+1} \leftarrow \boldsymbol{\mu}^l$
end

5.5.3 Global convergence

Algorithm SLA is cast within an existing SAO framework [11]; able to enforce global convergence through one of three mechanisms, namely:

1. Conservative convex separable Hessian approximations [78]. Conservatism was first proposed by Svanberg [62] and exploits a highly intuitive, but mathematically rigorous feature of approximations that prevents "large" uncontrolled steps. In [78] it was demonstrated that not all approximations need to be conservative as Svanberg originally proposed, but only active or violated constraints.

2. A trust-region method with a non-linear acceptance filter, described in [79, 80, 81, 82]. Trust region acceptance filters are today arguably considered the state-of-the-art in SQP-like methods, in all probability largely due to the fact that acceptance filters circumvent the need for troublesome merit filters in SQP, which rely on penalty parameters.

3. Filtered conservatism, which attempts to exploit the salient features of both conser-

vatism as popularized by Svanberg, and the trust-region method with a non-linear acceptance filter proposed by Fletcher and his co-workers [1].

A detailed comparison of all three global convergence mechanisms within an SAO framework can be found in [1]. However, since none of the problems in our numerical testing invoked global convergence for all the algorithms tested, we do not further elaborate on global convergence herein. For the sake of clarity and brevity, we instead refer the interested reader to the cited literature.

5.6 A primal-dual active-set strategy

For large-scale applications, active-set strategies may be prohibitively expensive, especially if the active-set changes many times per iteration k; requiring the linear system in (5.26) to be solved multiple times. To ameliorate traditional active-set difficulties, we make use of the efficient primal-dual active-set strategy proposed by Hintermüller *et al.* [77], which is ideally suited for the SQP subproblems that we exploit herein. Although far more robust active-set strategies exist, `SLA` is targeted at large-scale optimization problems, hence our choice for an efficient active-set strategy.

Equality constraints are always included in the active-set, while for inequality constraints, the primal-dual active-set strategy is given by [77]

$$\mathcal{J}^l = \begin{cases} j: & j = 1, \ldots, m', \\ j: \ \mu_j^k + f_j^k + \Psi_j^l > 0, & j = m'+1, \ldots, m, \end{cases} \tag{5.28}$$

where $\mathcal{J}^l$ contains the indices of the m_a active constraints at an iteration point l.

Following the dual update in (5.26), we remove constraints from the active-set $\mathcal{J}^l$ if the associated dual variable

$$\mu_j^* < 0, \quad j = m'+1, \ldots, m, \tag{5.29}$$

since we incorrectly assumed the respective dual variable resided in $(\mathcal{R}^m)^+$. Hence, we solve for the dual variables (5.26) with the updated active-set $\mathcal{J}^{l*}$, while noting that $\mathcal{J}^{l*} \subset \mathcal{J}^l$, where

$$\mathcal{J}^{l*} = \begin{cases} j: & j = 1, \ldots, m', \\ j: \ \mu_j^* \geq 0, & j = m'+1, \ldots, m, \end{cases} \tag{5.30}$$

thereby ensuring that all dual variables solved for in (5.26) are in $(\mathcal{R}^m)^+$, i.e. $\mu_j^l \geq 0, \ j = m'+1, \ldots, m$.

The operations in (5.26) are performed at most twice per subproblem iteration l, if the active-set was incorrectly identified in (5.28). However, our numerical testing indicates the active-set is often correctly identified in (5.28), such that $\mathcal{J}^l = \mathcal{J}^{l*}$, resulting in the dual variables (5.26) only being updated once per subproblem iteration l. The primal variables (5.27) require only a single update, following the final update for the dual variables. Obviously, no additional objective, constraint or first-order information is required in our proposed active-set strategy.

If an inequality constraint is inactive and not part of the active-set, then

$$\mu_j^l = 0, \quad j \notin \mathcal{J}^{l*}, \tag{5.31}$$

ensuring that complementary slackness is satisfied.

5.7 Numerical experiments

Herein, we demonstrate the efficacy of `SLA` for selected general and structural problems. For the structural problems, we additionally consider large-scale problems where $m \ll n$ and $m \approx n$. We compare `SLA` to the Falk dual [70], which is efficient when $m \ll n$, as well as to the state-of-the-art Galahad `LSQP` solver [51], since SQP methods are generally more efficient than pure dual methods when $m \approx n$. It is important to note that `LSQP` is *tailor-made* for *separable* approximations, accepting only *diagonal* Hessian information. The QP form in (5.23) is used to construct the approximate Lagrangian for `SLA`, while we use a pure dual statement for the Falk dual given in [71], with all solvers exploiting either the spherical diagonal approximation (SPH-QDR), or the quadratic approximation to the reciprocal approximation (T2:R).

The Falk dual is solved using the popular quasi-Newton `l-BFGS-b` solver [4], with default settings[5], except for $factr = 1$ and $pgtol = 10^{-10}$ (see [4] for the notations used here), since it is critical that the dual problem be solved to high-precision. Furthermore, we bound the Falk dual [6, 83] as a form of relaxation for ill-posed subproblems; a technique we also exploit in `SLA`. To update the primal variables, `SLA` uses the closed-form expression in (5.27), which is almost identical to the primal expression for the Falk dual[6] [71]; but whereas the Falk dual requires an iterative solver, we make use of direct linear solvers for the positive-definite dual system in (5.26). For problems with a dense constraint Jacobian, the LAPACK routine `dposv` [84] was used, together with the BLAS [85] library for matrix-matrix and matrix-vector operations. If the constraint Jacobian was sufficiently sparse, the Pardiso solver [86, 87, 88] in conjunction with the SPARSKIT library [89] was used to solve the subproblems. Default solver settings are enforced for Pardiso, using Cholesky factorization for a positive-definite matrix and no factorization reordering.

We now introduce the maximum constraint violation h, the Euclidean norm of the KKT conditions $\mathcal{K}$, the Euclidean primal norm $\|\mathbf{x} - \mathbf{x}^k\|_2$ and the number of iterations required for convergence, N_f. Problem execution was terminated when $h \leq 10^{-4}$, and either $\mathcal{K} \leq 10^{-4}$ or $\|\mathbf{x} - \mathbf{x}^k\|_2 \leq 10^{-5}$ was met. A move limit of 2×10^{-1} was used for all solvers, together with a maximum outer iteration count of $k = 500$. The dual variables were bound by a maximum value of 10^{12} to prevent numerical instabilities and initialized such that $\boldsymbol{\mu}^0 = 0$. The superscript * is used to denote the final solution values after termination of the problem, while '—' denotes that the maximum iteration count was reached without convergence.

[5] Herein, we will simply refer to the dual of Falk, solved using `l-BFGS-b`, as the "Falk dual". Note that `l-BFGS-b` is obviously not the only candidate for solving the resulting dual statement, and some authors claim superiority of conjugate-gradient (CG) methods able to accommodate bound constraints over `l-BFGS-b`, for presentable sets of test problems. Whether these methods will indeed provide a noticeable benefit for structural optimization remains to be seen, and we hope to report on this in the near future.

[6] The difference being in the denominator of (5.27). See the primal update in [71] for more details.

To solve for the test problems, we used double precision FORTRAN f90 under Ubuntu 18.04, compiled with gfortran-9 and the -O3 optimization flag invoked. All test problems were executed on a single quad-core Intel Xeon-4112 CPU, paired with 128 GB of 2666 MHz DDR4 system memory.

5.7.1 General and structural test problems

Table 5.1: The general and structural test problems. Note 1: The Hock and Schittkowski test set. Note 2: This is a generalization of Svanberg's cantilever.

#	Problem name	Approx.	n	m	Ref.	Notes
1	HS-6	SPH-QDR	2	1	[64]	1
2	HS-7	SPH-QDR	2	1	[64]	1
3	HS-35	SPH-QDR	3	1	[64]	1
4	HS-36	SPH-QDR	3	1	[64]	1
5	HS-43	SPH-QDR	4	3	[64]	1
6	HS-48	SPH-QDR	5	2	[64]	1
7	HS-49	SPH-QDR	5	2	[64]	1
8	HS-50	SPH-QDR	5	3	[64]	1
9	HS-76	SPH-QDR	4	3	[64]	1
10	HS-80	SPH-QDR	5	3	[64]	1
11	HS-111	SPH-QDR	10	3	[64]	1
12	HS-112	SPH-QDR	10	3	[64]	1
13	Svanberg's cantilever	SPH-QDR	5	1	[8]	
14	Toropov's cantilever	T2:R	1024	1	[11, 58]	2
15	Svanberg's snake problem	SPH-QDR	30	41	[62]	
16	Svanberg's first non-convex problem	SPH-QDR	200	2	[59]	
17	Svanberg's second non-convex problem	SPH-QDR	200	2	[59]	
18	Two-bar truss	SPH-QDR	2	2	[71]	
19	Fleury's weight minimization problem	T2:R	1000	2	[61]	
20	Vanderplaats' cantilever #1	T2:R	200	201	[60]	
21	Vanderplaats' cantilever #2	T2:R	200	200	[60]	
22	Vanderplaats' cantilever #3	T2:R	200	200	[60]	

We present the details of selected general and structural problems in Table 5.1, followed by the respective numerical results in Table 5.4. The results are in-line with expectations, where solver performance fluctuates across different problem types, highlighting the no-free-lunch (NFL) theorems for optimization [65]. Despite `SLA` and `LSQP` solving QP-like subproblems, compared to the pure dual statement of Falk, all three solvers perform similarly across the majority of problems. Both `SLA` and `LSQP` converged for all the test problems, albeit that

LSQP prematurely terminated on the Euclidean norm for Svanberg's notoriously difficult snake problem.

5.7.2 Large-scale structural test problems

The primal-dual separability of SLA allows for extremely efficient solutions to large-scale problems, demonstrated herein for cases when $m \ll n$ and $m \approx n$. Despite SLA being well-suited for implementation on massively parallel computational devices, all results and timings were performed on a modest single quad-core CPU.

The large-scale problems considered arise from Table 5.1; namely the three Vanderplaats' problems, Fleury's weight minimization problem and Toropov's cantilever. Details of the problem dimensionality and the required solution time is given in Table 5.5, with the numerical results presented in Table 5.6. Approximately half of the constraints are active at the solution for the Vanderplaats' problems, while all the constraints are active for the remaining large-scale problems. We used identical problem termination criteria as before, while additionally enforcing a maximum problem runtime limit of 10^4 seconds for all solvers.

Since the start vector $\mathbf{x}^0$ has a large influence on the iteration path, we note that a value of $\mathbf{x}^0 = 1$ was chosen for the Vanderplaats' problems and Toropov's cantilever, with Fleury's weight minimization problem initialized at the prescribed point of $\mathbf{x}^0 = 10^{-5}$.

For the Vanderplaats' problems where $m \approx n$, SLA clearly requires orders of magnitude lower time-complexity compared to LSQP, which in turn requires orders of magnitude lower time-complexity compared to the Falk dual. SLA was able to solve all three Vanderplaats' problems, containing up to a hundred million primal variables and a hundred million constraints, in just a few minutes. The solution time is highly dependent on exploiting sparsity in the constraint Jacobian and on the choice of linear solver, hence we exploited the high-performance Pardiso library, which the Galahad LSQP solver also exploits. All three solvers use a compressed sparse row (CSR) representation for the constraint Jacobian in the Vanderplaats' problems.

For Toropov's cantilever and Fleury's weight minimization problem where $m \ll n$, SLA again requires the lowest time-complexity, followed by the Falk dual. This is unsurprising, since the dual space is of low dimensionality for these problems. However, the Falk dual requires many more primal updates (5.18) compared to SLA, which can be computationally expensive when the primal dimensionality is large.

5.7.3 Topology optimization

We consider a single topology optimization example, for which the Falk dual is well known to be highly efficient (although a simple golden section or bisection approach also suffices). The problem is only of academic interest herein, since only a single (volume) constraint is present, but we include this problem to demonstrate that the bound constraints in topology optimization pose no problem for the primal projection strategy proposed in Section 5.5. The example is the well-known minimum compliance MBB beam problem, described in the seminal paper by Sigmund [90].

We called the example with the input line `top(300,100,0.5,3.0,8.0)`. For this problem[7], we present results using

1. T2:R for both the objective function f_0 and the linear volume constraint function f_1, and
2. T2:R for the objective function f_0 and SPH-QDR for the linear[8] volume constraint function f_1.

We will denote the above "Cases #1 and #2" respectively in the following. Numerical results are depicted in Figure 5.1 and Table 5.2 for Case #1, and in Figure 5.2 and Table 5.3 for Case #2. In each case, we have terminated the algorithms after 100 iterations. The geometries found by the different algorithms are comparable, with the most notable observations being the high computational effort required by `LSQP` (as can be expected), and the slightly superior black-and-white fraction found by `SLA` for Case #2 (and to a lesser extent, Case #1).

(a) `SLA` (b) `LSQP` (c) Falk dual

Figure 5.1: The MBB beam topology optimization problem, using the T2:R approximation for both the objective and constraint functions (Case #1).

[7]Firstly, we note that the original SIMP formulation is used, and not the modified SIMP method, simply due to the fact that only the former is currently present in our FORTRAN optimization environment. Secondly, the no-free-lunch (NFL) theorems clearly imply that no given algorithm, and by implication, approximation, will outperform all other algorithms or approximations for an arbitrary test set. We comment on the aforementioned NFL theorems, for structural optimization, in Chapter 8.

[8]While the NFL theorems reign supreme, the exploitation of *a priori* knowledge about the physics and mathematical statement of functions is advantageous, if not desirable, for the efficient solution of a given problem. Again, we comment on this in Chapter 8.

Table 5.2: The results for the MBB beam topology optimization example, using the T2:R approximation for both the objective and constraint functions (Case #1).

Solver	f_0^*	h^*	$\mathcal{K}^*$	l^*	u^*	Solver time (s)
SLA	2.122×10^{2}	1.89×10^{-15}	1.71×10^{-1}	8173	10454	5.81×10^{-1}
LSQP	2.118×10^{2}	0.00×10^{0}	1.40×10^{-1}	7999	10417	8.73×10^{0}
Falk dual	2.116×10^{2}	0.00×10^{0}	1.32×10^{-1}	8055	10515	3.51×10^{-1}

(a) SLA (b) LSQP (c) Falk dual

Figure 5.2: The MBB beam topology optimization problem, using the T2:R approximation for the objective function and the SPH-QDR approximation for the constraint (Case #2).

Table 5.3: The results for the MBB beam topology optimization example, using the T2:R approximation for the objective function and the SPH-QDR approximation for the constraint (Case #2).

Solver	f_0^*	h^*	$\mathcal{K}^*$	l^*	u^*	Solver time (s)
SLA	2.111×10^{2}	0.00×10^{0}	1.65×10^{-2}	8169	11263	4.00×10^{-1}
LSQP	2.120×10^{2}	5.71×10^{-14}	7.66×10^{-2}	7990	10427	8.74×10^{0}
Falk dual	2.117×10^{2}	0.00×10^{0}	7.05×10^{-2}	8081	10498	4.21×10^{-1}

5.8 Conclusions and recommendations

We have proposed a separable Lagrangian algorithm (SLA) for very large-scale optimal structural design. SLA solves a sequence of separable and convex quadratic-like problems, with the salient feature of closed-form primal and dual variable updates. The primal variables are updated in an embarrassingly parallel fashion, while the dual variables require the solution to a positive-definite linear system, determined from an efficient primal-dual active-set strategy. Furthermore, SLA is able to capture both reciprocal and exponential-like behavior by using intervening variables for diagonal Hessian approximations; a desirable property in structural optimization. Despite only presenting two Hessian approximations herein, many other possibilities exist, including the quadratic approximations to the MMA and CONLIN

approximations, etc.

Large-scale tests demonstrate that `SLA` is a robust and efficient solver for both $m \ll n$ and $m \approx n$, in contrast to traditional SQP or pure dual statements, which traditionally prefer either $m \approx n$ or $m \ll n$ respectively. Structural problems involving hundreds of millions of primal variables n and constraints m are solved for by `SLA` in a few minutes, requiring orders of magnitude less time-complexity compared to the state-of-the-art Galahad `LSQP` solver and popular Falk dual.

Although we have demonstrated the efficacy of a CPU implementation of `SLA`, the closed-form primal and dual variable updates are ideal for implementation on massively parallel computational devices. The time-complexity required for large-scale problems can particularly benefit from said implementations; a result we demonstrate in Section 5.10. Additionally, in future work, we wish to solve problems where $m \gg n$, an example of which can be found in [91]. Lastly, although we have experimented with only a single primal-dual active-set strategy, many other possibilities exist. In the near future, we hope to demonstrate the effects that various active-set and primal projection strategies have on the performance of `SLA`.

5.9 Tables for numerical results

Table 5.4: Numerical results for the general and structural test problems. '—' indicates that the algorithm failed to terminate within the maximum allowed number of iterations $k = 500$.

	SLA				LSQP				Falk			
#	f_0^*	h^*	$\mathcal{K}^*$	N_f	f_0^*	h^*	$\mathcal{K}^*$	N_f	f_0^*	h^*	$\mathcal{K}^*$	N_f
1	3.724×10^{-13}	1.49×10^{-9}	2.20×10^{-5}	21	3.965×10^{-15}	4.63×10^{-11}	3.86×10^{-6}	26	1.227×10^{-11}	6.81×10^{-9}	4.69×10^{-5}	16
2	-1.732×10^{0}	8.47×10^{-9}	3.48×10^{-8}	7	-1.732×10^{0}	1.91×10^{-11}	6.27×10^{-6}	11	-1.732×10^{0}	6.60×10^{-7}	1.16×10^{-6}	15
3	1.111×10^{-1}	0.00×10^{0}	1.09×10^{-5}	11	1.111×10^{-1}	0.00×10^{0}	2.08×10^{-5}	13	1.111×10^{-1}	2.80×10^{-14}	1.57×10^{-6}	11
4	-3.300×10^{3}	0.00×10^{0}	0.00×10^{0}	5	-3.300×10^{3}	0.00×10^{0}	8.32×10^{-14}	4	-3.300×10^{3}	0.00×10^{0}	3.57×10^{-7}	4
5	-4.400×10^{1}	1.03×10^{-9}	5.27×10^{-5}	10	-4.400×10^{1}	9.51×10^{-12}	7.01×10^{-6}	9	-4.400×10^{1}	1.46×10^{-11}	2.97×10^{-5}	8
6	4.269×10^{-10}	8.88×10^{-16}	6.97×10^{-5}	13	7.974×10^{-10}	0.00×10^{0}	8.87×10^{-5}	11	7.974×10^{-10}	4.44×10^{-16}	8.87×10^{-5}	11
7	2.322×10^{-6}	8.88×10^{-16}	7.95×10^{-5}	43	2.043×10^{-6}	0.00×10^{0}	7.22×10^{-5}	42	2.043×10^{-6}	8.88×10^{-16}	7.22×10^{-5}	42
8	4.124×10^{-14}	0.00×10^{0}	4.26×10^{-6}	10	1.059×10^{-24}	8.88×10^{-16}	5.11×10^{-5}	14	3.218×10^{-12}	4.23×10^{-11}	6.63×10^{-6}	14
9	-4.682×10^{0}	0.00×10^{0}	2.38×10^{-7}	7	-4.682×10^{0}	0.00×10^{0}	8.85×10^{-6}	9	-4.682×10^{0}	0.00×10^{0}	1.52×10^{-5}	7
10	5.395×10^{-2}	1.22×10^{-9}	9.32×10^{-6}	6	5.395×10^{-2}	1.22×10^{-9}	9.32×10^{-6}	6	5.395×10^{-2}	1.17×10^{-9}	9.32×10^{-6}	6
11	-4.776×10^{1}	8.79×10^{-12}	9.73×10^{-5}	148	-4.776×10^{1}	1.04×10^{-8}	9.88×10^{-5}	155	-4.776×10^{1}	5.02×10^{-9}	9.88×10^{-5}	163
12	-4.776×10^{1}	4.44×10^{-16}	9.31×10^{-3}	205	-4.776×10^{1}	2.22×10^{-16}	7.33×10^{-3}	247	-4.776×10^{1}	1.59×10^{-10}	6.50×10^{-3}	241
13	1.340×10^{0}	2.82×10^{-7}	3.59×10^{-5}	12	1.340×10^{0}	2.82×10^{-7}	3.59×10^{-5}	12	1.340×10^{0}	0.00×10^{0}	8.34×10^{-5}	27
14	1.310×10^{0}	3.96×10^{-6}	1.10×10^{-7}	10	1.310×10^{0}	3.96×10^{-6}	1.10×10^{-7}	10	1.310×10^{0}	6.04×10^{-6}	6.22×10^{-6}	11
15	-1.002×10^{1}	2.95×10^{-7}	4.96×10^{-5}	45	4.970×10^{0}	3.89×10^{-11}	1.87×10^{0}	29	—	—	—	—
16	5.105×10^{1}	0.00×10^{0}	9.48×10^{-5}	197	5.105×10^{1}	0.00×10^{0}	8.51×10^{-5}	185	5.105×10^{1}	0.00×10^{0}	9.74×10^{-5}	183
17	-1.490×10^{2}	5.59×10^{-10}	9.82×10^{-5}	306	-1.490×10^{2}	3.98×10^{-10}	9.70×10^{-5}	308	-1.490×10^{2}	3.60×10^{-9}	9.58×10^{-5}	309
18	1.509×10^{0}	4.83×10^{-11}	1.66×10^{-5}	12	1.509×10^{0}	1.84×10^{-9}	8.67×10^{-5}	9	1.509×10^{0}	0.00×10^{0}	6.19×10^{-5}	6
19	9.500×10^{2}	7.28×10^{-12}	6.35×10^{-6}	22	9.500×10^{2}	4.06×10^{-9}	9.90×10^{-4}	33	9.500×10^{2}	0.00×10^{0}	3.71×10^{-8}	67
20	6.365×10^{4}	2.35×10^{-14}	6.26×10^{-5}	23	6.365×10^{4}	2.29×10^{-14}	6.12×10^{-5}	22	6.365×10^{4}	1.36×10^{-6}	6.94×10^{-5}	31
21	5.416×10^{4}	1.42×10^{-14}	1.72×10^{-8}	17	5.416×10^{4}	7.11×10^{-15}	5.11×10^{-7}	16	5.416×10^{4}	4.00×10^{-6}	1.87×10^{-4}	32
22	5.416×10^{4}	1.42×10^{-14}	1.72×10^{-8}	17	5.416×10^{4}	7.11×10^{-15}	5.11×10^{-7}	16	5.416×10^{4}	4.00×10^{-6}	1.87×10^{-4}	32

Table 5.5: The large-scale structural test problems, together with the solution time for the respective solvers. † indicates that an algorithm failed to terminate within the time limit of 10^4 seconds, while '—' indicates that an algorithm failed to converge within $k = 500$ iterations.

				Solver time (s)		
Number	Problem name	n	m	`SLA`	`LSQP`	Falk
L-1a	Vanderplaats' cantilever #1	10^4	10^4+1	1.16×10^{-1}	1.47×10^{0}	2.61×10^{2}
L-1b		10^5	10^5+1	9.81×10^{-1}	1.66×10^{1}	†
L-1c		10^6	10^6+1	8.69×10^{0}	5.25×10^{2}	†
L-1d		10^7	10^7+1	1.03×10^{2}	5.35×10^{3}	†
L-1e		10^8	10^8+1	1.03×10^{3}	†	†
L-2a	Vanderplaats' cantilever #2	10^4	10^4	6.17×10^{-2}	8.78×10^{-1}	2.56×10^{2}
L-2b		10^5	10^5	5.84×10^{-1}	1.20×10^{1}	†
L-2c		10^6	10^6	5.60×10^{0}	1.29×10^{2}	†
L-2d		10^7	10^7	7.88×10^{1}	1.37×10^{3}	†
L-2e		10^8	10^8	8.25×10^{2}	†	†
L-3a	Vanderplaats' cantilever #3	10^4	10^4	6.47×10^{-2}	8.76×10^{-1}	2.55×10^{2}
L-3b		10^5	10^5	5.88×10^{-1}	1.20×10^{1}	†
L-3c		10^6	10^6	5.57×10^{0}	1.28×10^{2}	†
L-3d		10^7	10^7	7.83×10^{1}	1.37×10^{3}	†
L-3e		10^8	10^8	8.24×10^{2}	†	†
L-4a	Fleury's weight minimization problem	10^5	2	1.82×10^{-1}	1.27×10^{1}	2.03×10^{0}
L-4b		10^6	2	1.49×10^{0}	1.48×10^{2}	—
L-4c		10^7	2	2.18×10^{1}	—	—
L-5a	Toropov's cantilever	10^6	1	8.96×10^{-1}	4.93×10^{1}	2.13×10^{1}
L-5b		10^7	1	8.72×10^{0}	1.27×10^{3}	9.57×10^{0}
L-5c		10^8	1	8.61×10^{1}	†	7.22×10^{1}

Table 5.6: Numerical results for the large-scale structural test problems. † indicates that the algorithm failed to terminate within the time limit of 10^4 seconds, while '—' indicates that the algorithm failed to terminate within the maximum allowed number of iterations $k = 500$.

	SLA				LSQP				Falk			
#	f_0^*	h^*	$\mathcal{K}^*$	N_f	f_0^*	h^*	$\mathcal{K}^*$	N_f	f_0^*	h^*	$\mathcal{K}^*$	N_f
L-1a	6.364×10^{4}	5.66×10^{-13}	4.32×10^{-5}	22	6.364×10^{4}	6.44×10^{-13}	4.67×10^{-5}	21	6.364×10^{4}	5.00×10^{-6}	9.10×10^{-5}	34
L-1b	6.364×10^{4}	1.38×10^{-11}	6.83×10^{-5}	21	6.364×10^{4}	1.62×10^{-11}	7.41×10^{-5}	20	†	†	†	†
L-1c	6.364×10^{4}	1.37×10^{-11}	2.16×10^{-5}	21	6.364×10^{4}	1.63×10^{-11}	2.35×10^{-5}	20	†	†	†	†
L-1d	6.364×10^{4}	6.57×10^{-5}	3.43×10^{-5}	20	6.364×10^{4}	4.07×10^{-10}	3.73×10^{-5}	19	†	†	†	†
L-1e	6.364×10^{4}	6.14×10^{-6}	1.01×10^{-5}	20	†	†	†	†	†	†	†	†
L-2a	5.372×10^{4}	2.01×10^{-9}	3.22×10^{-6}	19	5.372×10^{4}	1.38×10^{-9}	1.79×10^{-6}	18	5.372×10^{4}	2.24×10^{-5}	4.51×10^{-5}	38
L-2b	5.371×10^{4}	4.07×10^{-5}	4.28×10^{-5}	20	5.371×10^{4}	7.59×10^{-9}	2.13×10^{-7}	20	†	†	†	†
L-2c	5.371×10^{4}	1.46×10^{-5}	1.00×10^{-6}	21	5.371×10^{4}	4.74×10^{-5}	2.38×10^{-5}	20	†	†	†	†
L-2d	5.371×10^{4}	3.23×10^{-6}	1.75×10^{-8}	22	5.371×10^{4}	0.00×10^{0}	9.64×10^{-6}	21	†	†	†	†
L-2e	5.371×10^{4}	4.46×10^{-5}	5.74×10^{-9}	23	†	†	†	†	†	†	†	†
L-3a	5.372×10^{4}	2.01×10^{-9}	3.22×10^{-6}	19	5.372×10^{4}	1.38×10^{-9}	1.79×10^{-6}	18	5.372×10^{4}	2.24×10^{-5}	4.51×10^{-5}	38
L-3b	5.371×10^{4}	4.07×10^{-5}	4.28×10^{-5}	20	5.371×10^{4}	7.59×10^{-9}	2.13×10^{-7}	20	†	†	†	†
L-3c	5.371×10^{4}	1.46×10^{-5}	1.00×10^{-6}	21	5.371×10^{4}	4.74×10^{-5}	2.38×10^{-5}	20	†	†	†	†
L-3d	5.371×10^{4}	3.23×10^{-6}	1.75×10^{-8}	22	5.371×10^{4}	0.00×10^{0}	9.64×10^{-6}	21	†	†	†	†
L-3e	5.371×10^{4}	4.46×10^{-5}	5.74×10^{-9}	23	†	†	†	†	†	†	†	†
L-4a	9.500×10^{6}	0.00×10^{0}	1.42×10^{-6}	18	9.500×10^{6}	9.90×10^{-9}	2.00×10^{3}	27	9.500×10^{6}	8.06×10^{-5}	6.34×10^{1}	358
L-4b	9.500×10^{8}	4.32×10^{-9}	1.27×10^{-2}	16	9.500×10^{8}	2.02×10^{-9}	1.06×10^{-2}	22	—	—	—	—
L-4c	9.500×10^{10}	7.90×10^{-8}	4.42×10^{-1}	21	—	—	—	—	—	—	—	—
L-5a	1.310×10^{0}	5.14×10^{-5}	8.50×10^{-6}	12	1.310×10^{0}	3.98×10^{-6}	5.03×10^{-7}	10	1.310×10^{0}	3.44×10^{-5}	4.83×10^{-6}	27
L-5b	1.319×10^{0}	2.95×10^{-5}	3.00×10^{-5}	9	1.310×10^{0}	1.08×10^{-7}	1.79×10^{-7}	12	2.188×10^{0}	0.00×10^{0}	9.87×10^{-5}	6
L-5c	1.322×10^{0}	6.54×10^{-6}	4.58×10^{-5}	9	†	†	†	†	2.184×10^{0}	0.00×10^{0}	3.12×10^{-5}	6

5.10 GPGPU implementation

As previously mentioned, the closed-form primal and dual variable expressions of `SLA` are well-suited for GPGPUs. Herein, we investigate the performance of a GPGPU implementation of `SLA`, within our existing SAOi framework [3]. Our GPGPU implementation of `SLA` is specifically for problems with a dense constraint Jacobian, since no tested GPGPU sparse solvers outperformed the Pardiso solver used in Section 5.7, especially for the large-scale Vanderplaats' problems considered in this chapter. It must be noted that the Vanderplaat's problems tested in Table 5.6 are extremely sparse. Instead, we test large-scale dense implementations of Vanderplaat's #2 and #3 that do not exploit sparsity in the Jacobian. The Vanderplaat's problems are chosen since problem dimensionality is easily scaled and $n \approxeq m$, the latter ensuring a large linear system can be formed in (5.26). The larger the linear system in (5.26), the more potential for GPGPU solver acceleration. Vanderplaat's #1 was not included in our results due to excessive memory requirements on our 11 GB GPGPU; a consequence of using a Cholesky factorization direct linear solver.

All GPGPU timings include host-to-device and device-to-host data transfers. Since we focus on a dense constraint Jacobian, the data transfer size may be in the order of gigabytes per subproblem. In real-world 'black-box' simulations, subproblem data will need to be loaded onto the GPGPU, hence our decision to include the data transfers in our timings. As far as possible, we implemented GPGPU BLAS operations using FORTRAN wrappers for the high-performance MAGMA library [92]. These largely consist of matrix-matrix and matrix-vectors operations, which are ideal for GPGPU implementation. MAGMA, as opposed to the vendor CUDA [68] BLAS library, was specifically selected for the hybrid CPU-GPGPU Cholesky factorization routine `dposv_gpu` [93]. The interested reader is referred to [93] for a detailed explanation of `dposv_gpu` and comparisons to other high-performance BLAS libraries, including CUDA and the Intel MKL library. For the GPGPU accelerated `SLA` implementation `SLA-GPGPU`, we use an identical test platform to that found in Section 6.7, while the CPU implementation uses the test platform in Section 5.7. All symbols and solver settings are identical to those outlined in Section 5.7.

Numerical results are presented in Table 5.7, where t_{GPU} and t_{CPU} denote the subproblem time required for the GPGPU and CPU implementations respectively. As expected, Table 5.7 indicates that the benefit of using a GPGPU increases with problem dimensionality, provided sufficient GPGPU memory is available. Figure 5.3 clearly shows a linear increase in $t_{\text{CPU}}/t_{\text{GPU}}$ with problem dimensionality; an encouraging result with the ever increasing memory available on GPGPUs. Despite the Jacobian host-to-device data transfer cost increasing with problem dimensionality, the GPGPU efficiently performs the expensive operations required to solve and construct the linear system in (5.26), thereby reducing the overall subproblem solution time.

Table 5.7: Numerical results for dense representations of the Vanderplaats' test problems [60], comparing the GPGPU and CPU implementations of `SLA`.

#	n	m	f_0^*	h^*	$\mathcal{K}^*$	N_f	t_{GPU} (s)	t_{CPU} (s)	$t_{\text{CPU}}/t_{\text{GPU}}$
2	1.0×10^4	1.0×10^4	5.372×10^4	2.01×10^{-9}	3.22×10^{-6}	19	1.11×10^2	3.35×10^2	3.02
	1.5×10^4	1.5×10^4	5.372×10^4	2.19×10^{-10}	4.19×10^{-7}	20	3.24×10^2	1.08×10^3	3.33
	2.0×10^4	2.0×10^4	5.372×10^4	2.61×10^{-6}	6.08×10^{-5}	19	5.86×10^2	2.18×10^3	3.72
	2.5×10^4	2.5×10^4	5.372×10^4	1.11×10^{-9}	6.03×10^{-7}	20	1.09×10^3	4.48×10^3	4.11
3	1.0×10^4	1.0×10^4	5.372×10^4	2.01×10^{-9}	3.22×10^{-6}	19	1.13×10^2	3.36×10^2	2.97
	1.5×10^4	1.5×10^4	5.372×10^4	2.19×10^{-10}	4.19×10^{-7}	20	3.25×10^2	1.10×10^3	3.38
	2.0×10^4	2.0×10^4	5.372×10^4	2.61×10^{-6}	6.08×10^{-5}	19	5.92×10^2	2.19×10^3	3.70
	2.5×10^4	2.5×10^4	5.372×10^4	1.11×10^{-9}	6.03×10^{-7}	20	1.08×10^3	4.49×10^3	4.16

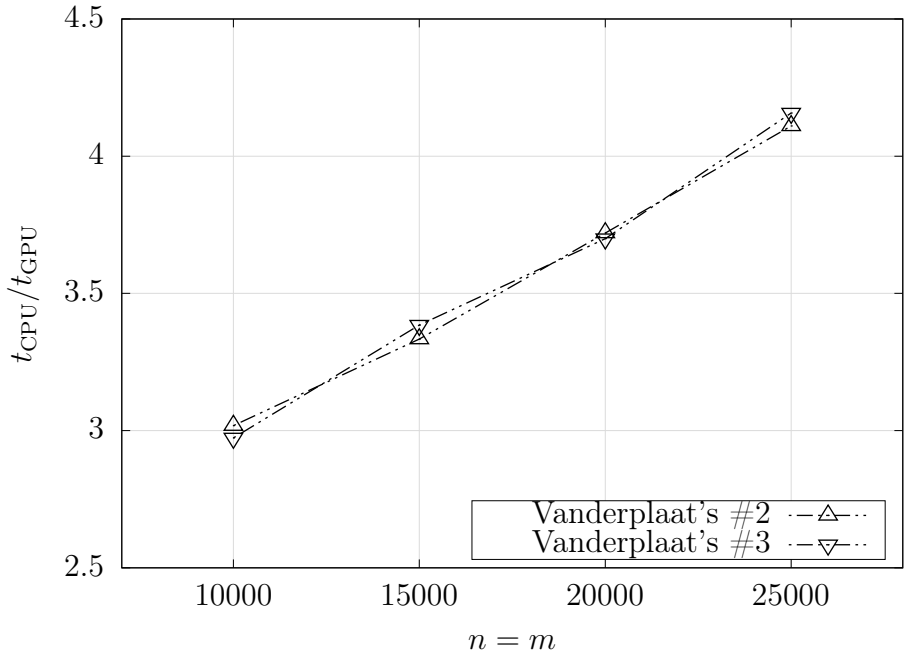


Figure 5.3: A comparison of the GPGPU and CPU implementations of `SLA`, for dense representations of the Vanderplaats' test problems [60].

Chapter 6

A deflation based low-rank singular value decomposition algorithm

The work presented here originates from a paper titled "A GPGPU accelerated, constrained and separable convex optimization approach for low-rank singular value decomposition" [94], which is under review at the time of writing. The paper is co-authored by Prof. Albert A. Groenwold.

6.1 Abstract

We propose a sequential approximate optimization approach for maximizing the well-known Rayleigh quotient, by solving a sequence of convex and separable approximate quadratic-like problems. The quadratic-like problems are cast into a sequential quadratic programming (SQP) formulation subject to a single constraint, with the salient feature of embarrassingly parallel primal and dual variable expressions. Our proposed implicitly restarted scaling strategy results in minimal SQP problems per singular-triplet, as well as the use of generic solver settings for different singular value distributions. The embarrassingly parallel primal and dual variable expressions are ideal for implementation on massively parallel computational devices, such as general purpose graphical compute units (GPGPUs), which we successfully demonstrate herein. Furthermore, our method is resilient to multiplicity and slowly decaying singular values, and requires *a priori* known constant memory.

To ensure that the primal and dual variable expressions remain embarrassingly parallel, successive singular values are solved for using a Schur deflation-based approach, hence we focus on low-rank decompositions of dense matrices. Our method demonstrates significant improvement in computation time compared to state-of-the-art Lanczos bidiagonalization methods, across an extensive set of large-scale test problems, in both CPU and GPGPU implementations. Numerical results show that both single and double precision variants of our method exploit GPGPUs to a far greater degree than MATLAB's GPGPU Lanczos `svds` algorithm.

6.2 Introduction

A real matrix $\mathbf{A} \in \mathcal{R}^{p\times n}$, which without loss of generality is either square, or has more rows p than columns n (i.e. $p \geq n$), can be rank-σ approximated by means of the Eckart-Young-Mirsky theorem [25], as

$$\mathbf{A}_\sigma = \mathbf{W}\mathbf{S}\mathbf{V}^{\mathrm{T}}, \tag{6.1}$$

where $\mathbf{A}_\sigma$ is

$$\min_{\mathbf{A}_\sigma} \|\mathbf{A} - \mathbf{A}_\sigma\|_F, \tag{6.2}$$

and $\|\cdot\|_F$ denotes the Frobenius norm. $\mathbf{W}$ is a $p \times \sigma$ orthonormal matrix, whose columns are the left-singular vectors of $\mathbf{A}$, $\mathbf{S}$ is a $\sigma \times \sigma$ square diagonal matrix containing the descending singular values, and $\mathbf{V}$ is an $n \times \sigma$ orthonormal matrix whose columns are the right-singular vectors of $\mathbf{A}$ [23, 25].

For big-data applications it is often times not feasible, nor desired, that the full decomposition be computed. Notwithstanding that numerous applications require a low-rank approximation of $\mathbf{A}$ [29], the full-rank decomposition is computationally expensive, with the best known algorithm requiring $\mathcal{O}(4p^2n + 22n^3)$ floating-point operations (flops) [23].

Recent randomized methods, which are based on Krylov subspaces and use some variation of the Arnoldi or Lanczos bidiagonalization method, have become popular for performing low-rank decompositions [29, 30, 95]. Although these methods are some of the most fruitful developments in eigenvalue and eigenvector extraction, methods exploiting Krylov subspaces do contain inherent shortcomings. They suffer from numerical instability due to propagated round-off errors and struggle to converge if the singular values of $\mathbf{A}$ contain multiplicity or decay slowly[1] [29, 30, 98], thereby requiring many iterations that may introduce prohibitively expensive memory and computational demands [98]. Techniques to ameliorate these issues include preconditioning and restarting; but as effective as they have become, do not completely resolve the underlying issues inherent to methods exploiting Krylov subspaces.

The varying memory requirements of Krylov methods make implementation on today's GPGPUs unreliable, since although $\mathbf{A}$ may fit into GPGPU memory, Krylov methods may require more than the available memory depending on the singular value distribution of $\mathbf{A}$. In practice, the singular value distribution is not known in advance, resulting in possible failure of GPGPU Krylov methods to properly perform the singular value decomposition. Hence, it is of *critical* importance that a method requires *constant* memory for in-core GPGPU implementation, which the Lanczos methods fail at, since memory on today's devices is somewhat limited. Although additional memory can be reserved on the device for Lanczos methods, the maximum allowable problem size may be significantly reduced, especially if the singular values contain multiplicity or decay slowly[2]. Notwithstanding their known memory challenges, Lanczos methods are still able to exploit parallel computation [99, 100, 101]; albeit to a lesser degree than our proposed method[3]. Literature is scarce on GPGPU implementations of Lanczos methods, most likely due to the aforementioned shortcomings, as well as

[1] Singular values that are poorly separated often occur in practical applications, with examples given in References [96] and [97].

[2] We strictly focus on in-core computation.

[3] See the results of Section 6.7 in comparison to [99] and the parallel assessment therein.

the $\sigma \times \sigma$ full-rank SVD performed per iteration [29, 30]. The $\sigma \times \sigma$ full-rank SVD required by Lanczos methods can be prohibitively expensive and difficult to effectively implement on GPGPUs, especially if σ is small and many Lanczos iterations are required.

Herein, we introduce a constrained convex optimization approach to maximize the Rayleigh quotient, which requires *constant* memory and can *entirely* be performed in-core on a GPGPU. Since both the primal and dual variable updates are embarrassingly parallel and require simple matrix-vector operations, our proposed method is ideal for GPGPU implementation. Furthermore, our method is relatively insensitive to multiplicity or slowly decaying singular values, which we demonstrate in our numerical results. Although similar convex optimization approaches exist [102, 103], they often involve solving a sequence of *unconstrained* convex problems in $\mathcal{R}^n$. For large dimensionalities of $\mathcal{R}^n$, solving the convex problems may be expensive, especially if the solution requires serial operations. In contrast, our method takes inspiration from structural optimization and the dual of Falk [70], whereby constrained convex problems are solved for in the so-called *dual space*, using approximate second-order Hessian information. Under the assumption that the strictly convex problems are separable, the primal variables are updated using analytical expressions, once the dual variables are solved for. Our problem formulation is subject to only a single equality constraint, resulting in a one-dimensional dual space; hence the dual variable updates, as well as the primal updates, are computationally efficient.

Consider a nonlinear problem $\mathcal{P}_{\text{NLP}}$ of the form

$$\begin{aligned} \min_{\mathbf{x}} \; & f(\mathbf{x}) \\ \text{subject to} \; & g(\mathbf{x}) = 0 \\ & \check{x}_i \leq x_i \leq \hat{x}_i, \qquad i = 1, 2, \ldots, n, \end{aligned} \tag{6.3}$$

where $f(\mathbf{x}) : \mathcal{R}^n \to \mathcal{R}$ denotes an objective function and $g(\mathbf{x}) : \mathcal{R}^n \to \mathcal{R}$ a single equality constraint function, both of which depend upon the n primal variables $\mathbf{x} = \{x_1, x_2, \cdots, x_n\}^{\mathrm{T}} \in \mathcal{X} \subset \mathcal{R}^n$. The lower and upper bounds of a variable x_i are denoted by $\check{x}_i$ and $\hat{x}_i$ respectively, ensuring that $\mathcal{X}$ is bound and closed.

Assuming that (6.3) is at least once continuously differentiable, sequential approximate optimization (SAO) may be used to solve for (6.3), since it results in a highly efficient SVD algorithm with embarrassingly primal and dual variable updates. SAO, which is arguably state-of-the-art in structural optimization, solves an iterative sequence of strictly *convex* and *separable* quadratic-like approximate problems $\mathcal{P}_{\text{P}}[k]$, given by

$$\begin{aligned} \min_{\mathbf{x}} \; & \tilde{f}^k(\mathbf{x}) \\ \text{subject to} \; & \tilde{g}^k(\mathbf{x}) = 0, \\ & \check{x}_i^k \leq x_i \leq \hat{x}_i^k, \qquad i = 1, \ldots, n, \end{aligned} \tag{6.4}$$

where $k = 0, 1, \ldots$ denotes the iteration count of the continuous approximate primal problem $\mathcal{P}_{\text{P}}[k]$. For a given iteration k, $\mathcal{P}_{\text{P}}[k]$ contains n unknown primal variables, a single equality constraint and $2n$ bound constraints, the latter requiring minimal computational effort, as we demonstrate in sections to come.

The separability in SAO arises from the use of approximate diagonal Hessian (second-order) information, since exact Hessian information is often prohibitively expensive for large-scale problems in structural optimization. The exact Hessian storage alone is of $\mathcal{O}(n^2)$, which is not acceptable for in-core computation on the memory limited devices currently available. In contrast, approximate diagonal Hessian information requires $\mathcal{O}(n)$ storage, and can be updated from the analytical solution of a simple first-order Taylor series, thereby requiring minimal computational and memory costs. Popular algorithms that exploit separable information include the convex linearization algorithm (CONLIN) of Fleury and Braibant [10], as well as Svanberg's generalization of CONLIN, namely the method of moving asymptotes (MMA) [8, 9].

To condition the convex optimization problem formed from $\mathcal{P}_{\mathrm{P}}[k]$, we make use of an implicitly restarted scaling strategy, which relies on information from previously solved for singular values. As is commonly known, scaling of optimization problems is of utmost importance, as so enticingly mentioned by Papalambros and Wilde [34] (verbatim): "In any case, one cannot overemphasize that *scaling is the single most important, but simplest, reason that can make the difference between success and failure of a design optimization algorithm."* The proposed scaling strategy works remarkably well, by reducing the number of approximate problems (k) required for each singular-triplet, while allowing generic solver settings across singular values of differing magnitudes.

Our method has fixed, *a priori* known memory requirements and is resilient in the presence of multiplicity or slowly decaying singular values. The GPGPU implementation of our algorithm is memory bandwidth limited on today's devices; an area rapidly improving with hardware advances. Additionally, we have transparent flexibility between speed and accuracy, by modifying the convergence criterion of the optimization algorithm used.

The contributions of this study are summarized as follows: We propose a constrained optimization approach for low-rank SVD, which contains embarrassingly parallel primal and dual variable updates. Hence, the algorithm is ideal for efficient GPGPU implementation, which we successfully demonstrate herein. To condition the separable approximate convex optimization problems and allow the use of generic solver settings, we use a novel implicitly restarted scaling strategy, which results in only a small number of approximate convex problems per singular triplet. Finally, we present large-scale numerical results for both CPU and GPGPU implementations of our algorithm, where we demonstrate competitiveness against state-of-the-art Lanczos methods.

Our paper is arranged as follows: In Section 6.3 we introduce the Rayleigh quotient maximization problem, which is our optimization problem of choice for a desired singular-triplet. In Section 6.4, the approximate convex functions are introduced, together with a sequential quadratic program (SQP) constructed from the approximate functions. The SQP is solved using the closed-form primal and dual updates derived in Section 6.5, before introducing the implicitly restarted scaling strategy in Section 6.6. Numerical results for artificial and real-world datasets are presented in Section 6.7, where we compare our method to state-of-the-art Lanczos methods. Finally, recommendations and conclusions are discussed in Section 6.8.

6.3 Problem formulation

We depart with the well-known eigenvector and eigenvalue relationship [26], given by

$$\hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}\mathbf{x}_r = \lambda_r \mathbf{x}_r. \tag{6.5}$$

Multiplying by $\mathbf{x}_r^{\mathrm{T}}$, we obtain

$$\mathbf{x}_r^{\mathrm{T}}\hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}\mathbf{x}_r = \mathbf{x}_r^{\mathrm{T}}\lambda_r \mathbf{x}_r. \tag{6.6}$$

We note that $\mathbf{x}_r^{\mathrm{T}}\mathbf{x}_r = \|\mathbf{x}_r\|_2^2$ and λ_r are scalars and solve for the eigenvalue of the covariance structure

$$\lambda_r = \frac{\mathbf{x}_r^{\mathrm{T}}\hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}\mathbf{x}_r}{\|\mathbf{x}_r\|_2^2}, \tag{6.7}$$

which is the well-known Rayleigh quotient [27, 28]. The expression in (6.7) needs to be maximized to obtain the eigenvalue associated with the most variance, hence we *minimize* the unconstrained problem

$$\min_{\mathbf{x}_r} f(\mathbf{x}_r) = -\frac{\mathbf{x}_r^{\mathrm{T}}\hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}\mathbf{x}_r}{\|\mathbf{x}_r\|_2^2} \qquad r = 1, \ldots, \sigma, \tag{6.8}$$

where

$$\hat{\mathbf{A}} = \mathbf{A} - \sum_{s=1}^{r-1} \mathbf{A}\frac{\mathbf{x}_s^* \mathbf{x}_s^{*\mathrm{T}}}{\mathbf{x}_s^{*\mathrm{T}}\mathbf{x}_s^*}. \tag{6.9}$$

Although the problem in (6.8) may be solved for by a host of unconstrained techniques, including gradient descent, conjugate gradient or quasi-Newton methods; large problem sizes will result in $\mathcal{R}^n$ being a high-dimensional search space, which may be expensive for unconstrained methods searching in $\mathcal{R}^n$. Instead, we reformulate the problem, such that

$$\begin{aligned} \min_{\mathbf{x}_r} \; & f(\mathbf{x}_r) = -\mathbf{x}_r^{\mathrm{T}}\hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}\mathbf{x}_r \\ \text{subject to} \; & g(\mathbf{x}_r) = \|\mathbf{x}_r\|_2^2 - 1 = 0, \end{aligned} \tag{6.10}$$

which contains only a single equality constraint. In structural optimization, dual methods are proven to be highly efficient when the number of primal variables far exceeds the number of constraints, since the search space is confined to the dual space. If the explicitly expressed objective function and constraints are separable and strictly convex, it is well-known that closed-form analytical expressions exist for the primal variables, as demonstrated by Falk [70]. Indeed, the approximate convex functions we use throughout are both separable and convex, allowing for the efficient update of the primal variables. Traditionally, pure dual methods require the use of iterative solvers [52, 71], since the pure dual problem is 'piecewise quadratic' and second-order discontinuous [71, 76]. However, we demonstrated in Chapter 5 that a closed-form solution exists for both the primal and dual variables, should an SQP formulation be used instead of a pure dual statement, which we prove in sections to come. The closed-form updates are ideal for a GPGPU implementation, hence we prefer the SQP formulation to the pure dual statement, notwithstanding that the pure dual statement is still efficient for the problem in (6.10).

The gradients of (6.10), which are stationary at an eigenvector $\mathbf{x}_r^*$ [27] and required by the solver, are

$$\begin{aligned} \boldsymbol{\nabla} f(\mathbf{x}_r) &= -2\hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}\mathbf{x}_r, \\ \boldsymbol{\nabla} g(\mathbf{x}_r) &= 2\mathbf{x}_r, \end{aligned} \tag{6.11}$$

where $\boldsymbol{\nabla}$ represents the primal differential operator. The problem in (6.10) is convex in the unit sphere, hence the stationary points of (6.10) are unique to the desired eigenvector $\mathbf{x}_r^*$ when (6.10) is minimized [5, 27, 33]. The update in (6.9) performs a covariance-free Schur complement deflation technique on $\mathbf{A}$ [104], which preserves the positive semi-definite structure of the covariance matrix, while removing variance in the direction of the previously solved for eigenvectors. Deflation is a popular technique in principle component analysis (PCA), where the solution eigenvector, $\mathbf{x}_r^*$, contains non-redundant information about the previously found eigenvectors $\mathbf{x}_1^*, \ldots, \mathbf{x}_{r-1}^*$ [104]. However, if $\mathbf{A}$ is ill-conditioned and many modes are desired, round-off error can accumulate and cause accuracy issues. Herein, we focus only on low-rank decompositions, hence round-off errors have not been problematic in our numerical testing.

The corresponding eigenvector of $\hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}$ represents a right-singular vector, $\mathbf{v}_r$, of $\mathbf{A}$. Expressed in terms of the solution vector, we obtain

$$\mathbf{v}_r = \frac{\mathbf{x}_r^*}{\|\mathbf{x}_r^*\|_2}. \tag{6.12}$$

A left-singular vector, $\mathbf{w}_r$, is analytically determined using [99]

$$\mathbf{w}_r = \frac{\mathbf{A}\mathbf{v}_r}{s_r}, \tag{6.13}$$

where the scalar

$$s_r = \sqrt{-f(\mathbf{x}_r^*)} = \sqrt{\lambda_r}, \tag{6.14}$$

is the corresponding singular value. The eigenvalues of the symmetric covariance matrix in (6.10) are positive and real, which ensures that the singular values are real, i.e. $s_r \in \mathcal{R}$. Additionally, the eigenvalues of the covariance structures $\hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}$ and $\hat{\mathbf{A}}\hat{\mathbf{A}}^{\mathrm{T}}$ are the squares of their respective singular values [105].

Alternatively, the covariance structure $\hat{\mathbf{A}}\hat{\mathbf{A}}^{\mathrm{T}}$ may be used in (6.10) to solve for the left-singular vectors. We choose to solve for the right-singular vectors from our definition of $p \geq n$, since a smaller $n \times n$ covariance matrix $\hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}$ is formed. The modes may also be solved for in ascending order by maximizing (6.10), but the smallest modes can be difficult or impossible to solve for due to numerical instabilities, especially since our formulation relies on the formation of $\hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}$, albeit implicitly. An important note: we never calculate the covariance structures in our function or gradient updates, since we note that

$$\mathbf{x}_r^{\mathrm{T}}\hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}\mathbf{x}_r = \left(\hat{\mathbf{A}}\mathbf{x}_r\right)^{\mathrm{T}}\left(\hat{\mathbf{A}}\mathbf{x}_r\right),$$

hence updating (6.10) requires one matrix-vector operation, followed by the inner product of the resulting vectors. Similarly, (6.11) can be computed from the products formed in (6.10). The covariance-free update, involving no matrix-matrix operations, significantly reduces the required computational cost.

6.4 Quadratic-like approximations

Indeed, the constrained convex problem in (6.10) may be solved for using SAO, whereby a sequence of approximate, separable convex subproblems are constructed and solved for. Since each singular-triplet is solved for as the extreme eigenvalue of $\hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}$, with further singular-triplets requiring deflation, we use $\mathbf{x} = \mathbf{x}_r$ for sake of brevity. The use of the underscore r, to denote singular information at a given index, will be made clear from the context throughout.

To construct the quadratic-like subproblems, an approximate objective function $\tilde{f}(\mathbf{x}^k)$ and constraint function $\tilde{g}(\mathbf{x}^k)$ are constructed using a second-order incomplete series expansion (ISE) [2], around a primal iterate $\mathbf{x}^k$

$$\begin{aligned} \tilde{f}^k(\mathbf{x}) &= f(\mathbf{x}^k) + (\boldsymbol{\nabla} f(\mathbf{x}^k))^{\mathrm{T}}\mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}\mathbf{C}^k\mathbf{s}, \\ \tilde{g}^k(\mathbf{x}) &= \mathbf{g}(\mathbf{x}^k) + (\boldsymbol{\nabla}\mathbf{g}(\mathbf{x}^k))^{\mathrm{T}}\mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}(2\mathbf{I})\,\mathbf{s}. \end{aligned} \tag{6.15}$$

Here, $\mathbf{s} = (\mathbf{x} - \mathbf{x}^k)$ and $\mathbf{C}^k$ is diagonal matrix containing approximate second-order information of the objective function. Since the single constraint is completely separable, we use exact second-order information given by $2\mathbf{I}$, where $\mathbf{I} \in \mathcal{R}^{n\times n}$ is the identity matrix.

For the sake of brevity and clarity, the abbreviated notation

$$\begin{aligned} f^k &= f(\mathbf{x}^k), \quad g^k = g(\mathbf{x}^k), \\ \left(\frac{\partial f}{\partial x_i}\right)^k &= \frac{\partial f(\mathbf{x}^k)}{\partial x_i}, \quad \left(\frac{\partial g}{\partial x_i}\right)^k = \frac{\partial g(\mathbf{x}^k)}{\partial x_i}, \\ \boldsymbol{\nabla} f^k &= \left[\left(\frac{\partial f}{\partial x_1}\right)^k, \ldots, \left(\frac{\partial f}{\partial x_n}\right)^k\right]^{\mathrm{T}}, \\ \boldsymbol{\nabla} g^k &= \left[\left(\frac{\partial g}{\partial x_1}\right)^k, \ldots, \left(\frac{\partial g}{\partial x_n}\right)^k\right]^{\mathrm{T}}, \end{aligned}$$

will be used throughout, allowing the approximate functions to be expressed as

$$\begin{aligned} \tilde{f}^k(\mathbf{x}) &= f^k + \boldsymbol{\nabla} f^{k\mathrm{T}}\mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}\mathbf{C}^k\mathbf{s}, \\ \tilde{g}^k(\mathbf{x}) &= g^k + \boldsymbol{\nabla} g^{k\mathrm{T}}\mathbf{s} + \mathbf{s}^{\mathrm{T}}\mathbf{I}\mathbf{s}. \end{aligned} \tag{6.16}$$

Although SAO approaches for structural optimization problems commonly rely upon intervening variables, we will herein consider a Hessian approximation suitable for general optimization problems, namely a spherical quadratic approximation. This approximation is presented by Snyman and Hay in [54], whereby the second-order curvature is chosen such that the real function at the previous iterate, f^{k-1}, is equal to approximate function value at the previous iterate, $\tilde{f}^{k-1}$. Hence, we enforce

$$\tilde{f}(\mathbf{x}^{k-1}) = f(\mathbf{x}^{k-1}), \tag{6.17}$$

which requires the determination of the single unknown curvature c^k, given by

$$c^k = \frac{2[f^{k-1} - f^k - \boldsymbol{\nabla} f^{k\mathrm{T}}(\mathbf{x}^{k-1} - \mathbf{x}^k)]}{\|\mathbf{x}^{k-1} - \mathbf{x}^k\|_2^2}. \tag{6.18}$$

Since the approximation is the spherical approximation proposed by Snyman and Hay [54], the approximate objective function second-order information is given by

$$\mathbf{C}^k = c^k \mathbf{I}, \tag{6.19}$$

with alternative formulations for spherical approximations presented in [55]. For the first iteration, $k = 0$, a curvature of 10^1 is rather arbitrarily assumed, since no historical information is available. In our numerical testing, a more conservative initial curvature estimate generally resulted in fewer iterations k per singular-triplet.

To solve for the primal problem $\mathcal{P}_{\mathrm{NLP}}$, we transform the separable approximations (6.16) into convex, separable quadratic approximate programs $\mathcal{P}_{\mathrm{PQ}}[k]$, expressed as

□ *Quadratic approximate program* $\mathcal{P}_{\mathrm{PQ}}[k]$

$$\begin{aligned} \min_{\mathbf{x}} \ & \tilde{f}^k(\mathbf{x}) = f^k + \boldsymbol{\nabla} f^{k\mathrm{T}}\mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}\mathbf{Q}^k\mathbf{s} \\ \text{subject to} \ & \tilde{g}^k(\mathbf{x}) = g^k + \boldsymbol{\nabla} g^{k\mathrm{T}}\mathbf{s} = 0, \\ & \check{x}_i^k \leq x_i \leq \hat{x}_i^k, \qquad\qquad i = 1, 2, \ldots, n, \end{aligned} \tag{6.20}$$

where $\mathbf{Q}^k \in \mathcal{R}^{n\times n}$ is the Hessian of the approximate Lagrangian $\tilde{\mathcal{L}}^k$. The interested reader is referred to Etman *et al.* [52, 53] for more information regarding $\mathcal{P}_{\mathrm{PQ}}[k]$.

Since we exploit a sequential quadratic programming (SQP) formulation in (6.20), the Lagrange multipliers, or dual variables, at the previous iterate are used to construct $\mathbf{Q}^k$, along with exact second-order information for the single constraint, such that

$$q^k = \max(\epsilon > 0, c^k + 2\mu^k) \tag{6.21}$$

As with the primal variables, for sake of brevity we let $\mu = \mu_r$, since only a single dual variable is solved for per singular-triplet. In (6.21) we enforce convexity of (6.20) by selecting ϵ as a 'small' value of 10^{-6} throughout our numerical testing. Finally, the approximate diagonal Hessian of $\tilde{\mathcal{L}}^k$ is given by

$$\mathbf{Q}^k = q^k \times \mathbf{I}. \tag{6.22}$$

6.5 Embarrassingly parallel primal-dual updates

The convergence of SAO algorithms for convex problems is well-known, with but one example given in [7]. Notwithstanding that global convergence of SAO algorithms is not guaranteed without a suitable global enforcement mechanism, we are able to achieve local convergence throughout all our numerical testing, by using two iterations of the power method on a uniformly randomly distributed initial primal vector. This results in significantly faster

convergence rates, as opposed to enforcing global convergence through various tested mechanisms; such as the conservatism in [78], a trust-region method with a non-linear acceptance filter [79, 80, 81, 82] or filtered conservatism [1].

Although the method outlined in this section is similar to solving for the first singular-triplet in Chapter 7, the use of deflation, as well as the lack of a global enforcement mechanism, results in a significantly different iteration path. Furthermore, our method in Chapter 7 is more expensive, both in terms of computational complexity and memory requirements, than the current method for solving further modes.

As previously mentioned, dual SAO methods are popular for problems containing far more primal variables than constraints. Indeed, the problem in (6.20) is an ideal candidate for dual methods, since only a single constraint is present. One such popular algorithm, which relies upon separable and convex objective and constraint functions, is the dual of Falk [70]. A desirable property of the Falk dual is the closed-form, embarrassingly parallel primal minimizers, which are updated following the solution of the dual variables. In further work of ours on the Falk dual [69], we demonstrated that both the primal and dual variables can be updated using closed-form expressions, albeit that the dual updates are not embarrassingly parallel. This produced a highly efficient algorithm for large-scale structural problems, which we apply to a more general optimization problem herein. The algorithm becomes a feasible descent method for equality constrained problems, with closed-form primal and dual variable updates, requiring minimal subproblem solution effort. In this section, we outline a variant of the algorithm in [69], which achieves greater levels of efficiency under certain assumptions of the problem in (6.20). One such efficiency increase is the embarrassingly parallel dual updates, resulting from the presence of a single constraint. The second is the relaxation of the primal projection strategy in [69] to handle primal variables residing on the bounds of $\mathcal{X}$, since the solution $\mathbf{x}^*$ will always be an interior-point of $\mathcal{X}$, as shown in what is to follow.

We depart with the popular Lagrangian for the quadratic program $\mathcal{P}_{\mathrm{P}}[k]$ in (6.20), given by

$$\tilde{\mathcal{L}}^k(\mathbf{x}, \mu) = \tilde{f}^k(\mathbf{x}) + \tilde{g}^k(\mathbf{x})^{\mathrm{T}}\mu, \quad \mathbf{x} \in \mathcal{X} \subset \mathcal{R}^n. \tag{6.23}$$

The dual of Falk [70] is an ideal candidate to solve for (6.23), since both the objective and constraint functions are separable and convex. The dual, or auxiliary, function of Falk [70], is defined as

$$\begin{aligned} \tilde{\gamma}^k(\mu) = \quad & \min_{\mathbf{x}} \tilde{\mathcal{L}}^k(\mathbf{x}, \mu), \\ \text{subject to} \quad & \mathbf{x} \in \mathcal{X}, \end{aligned} \tag{6.24}$$

where, as previously mentioned, the set $\mathcal{X}$ is bound and closed, containing the respective upper and lower bounds of $\mathbf{x}$ at an iteration k. Since we do not make use of any move limit strategy herein, the upper and lower bounds on $\mathbf{x}$ are constant for all k, hence $\mathcal{X}^k = \mathcal{X}\ \forall\ k$. The dual *problem* of Falk may then be expressed as

$$\begin{aligned} & \max_{\mu} \tilde{\gamma}^k(\mu) \\ \text{subject to} \quad & \mathbf{x}(\mu) \in \mathcal{X}, \end{aligned} \tag{6.25}$$

since the primal minimizer $\mathbf{x}(\mu)$ is a function of μ. This can be seen by noting that the primal minimizer of $\tilde{\mathcal{L}}^k(\mathbf{x}, \mu)$, $\mathbf{x} \in \mathcal{X}$ depends on μ. Indeed, the dual problem may be expressed solely as a function of the single dual variable, such that

$$\begin{aligned} \max_{\mu} \ & \tilde{\gamma}^k(\mu) = \tilde{f}^k(\mathbf{x}(\mu)) + \tilde{g}^k(\mathbf{x}(\mu))^{\mathrm{T}}\mu \\ \text{subject to} \quad & \mathbf{x}(\mu) \in \mathcal{X}. \end{aligned} \tag{6.26}$$

The result in (6.26) is particularly popular in structural optimization [71, 76], especially since $\mathbf{x}(\mu)$ is available as a closed-form expression, as we will show in what is to come. As we demonstrated in [69], if the solution $\mathbf{x}^* = \mathbf{x}(\mu^*)$ is an interior-point of $\mathcal{X}$, then the constraint $\mathbf{x}(\mu) \in \mathcal{X}$ may be relaxed and a simple bound-projection used instead. Since we choose $\hat{x}_i^k = -\check{x}_i^k = 1 \ \forall \ k$, the solution $\mathbf{x}(\mu^*)$ is always an interior-point of $\mathcal{X}$[4], allowing for an *unconstrained* dual problem, given by

$$\max_{\mu} \ \tilde{\gamma}^k(\mu) = \tilde{f}^k(\mathbf{x}(\mu)) + \tilde{g}^k(\mathbf{x}(\mu))^{\mathrm{T}}\mu. \tag{6.27}$$

The Karush-Kuhn-Tucker (KKT) system, derived from the first-order conditions of (6.27), will always be of full-rank because of the single active constraint, hence a unique primal-dual solution exists. Once continuously differentiating (6.27) with respect to μ and performing some matrix calculus[5], results in a closed-form dual update

$$\mu^{k+1} = \mu^* = \left(q^k g^k - \boldsymbol{\nabla} g^{k\mathrm{T}} \boldsymbol{\nabla} f^k\right) / \|\boldsymbol{\nabla} g^k\|_2^2, \tag{6.28}$$

with the inner-product $\boldsymbol{\nabla} g^{k\mathrm{T}} \boldsymbol{\nabla} f^k$ forming a scalar. Following the dual update in (6.28), the primal variables are updated as

$$\mathbf{x}(\mu^*) = \mathbf{x}^k - \left(\boldsymbol{\nabla} f^k + \mu^* \boldsymbol{\nabla} g^k\right) / q^k, \tag{6.29}$$

a result shown by Falk [70] and others [71, 76]. To ensure numerical stability, we project a primal variable $x_i(\mu^*)$ onto $[\check{x}_i, \hat{x}_i]$, such that

$$x_i^{k+1} = \begin{cases} x_i(\mu^*) & \text{if} \quad \check{x}_i < x_i(\mu^*) < \hat{x}_i \\ \hat{x}_i & \text{if} \quad x_i(\mu^*) \geq \hat{x}_i \\ \check{x}_i & \text{if} \quad x_i(\mu^*) \leq \check{x}_i \end{cases}, \quad i = 1, \ldots, n, \tag{6.30}$$

where $\hat{x}_i = -\check{x}_i = 1 \ \forall \ i$ in our numerical testing. To abbreviate (6.30), we shall use the projection operator $\Pi(\cdot)$, resulting in the primal update

$$\mathbf{x}^{k+1} = \Pi\left(\mathbf{x}(\mu^*)\right). \tag{6.31}$$

Both the dual and primal variable updates in (6.28) and (6.31) are ideal for implementation on GPGPUs, provided the problem size is sufficiently large, since both operations are closed-form and involve embarrassingly parallel vector-vector and matrix-vector operations.

[4] At the solution point, the constraint $\|\mathbf{x}(\mu^*)\|_2^2 = 1$ must be satisfied.

[5] From theory, the first-order conditions given in (6.28) may alternatively be obtained by evaluating the single constraint function at $g^k(\mathbf{x}(\mu^*)) = 0$.

Algorithm 3: Algorithm `saosvd-d` for rank-σ SVD

Initialize: $r = 1, \mathbf{x} \in \mathcal{R}^n, \mathbf{A} \in \mathcal{R}^{p\times n}, \mathbf{W} \in \mathcal{R}^{p\times\sigma}, \mathbf{S} \in \mathcal{R}^{\sigma\times\sigma}, \mathbf{V} \in \mathcal{R}^{n\times\sigma}$
repeat
  Construct $\hat{\mathbf{A}}$ with (6.9)
  k = 0
  repeat
    Update $f^k, \boldsymbol{\nabla} f^k, g^k, \boldsymbol{\nabla} g^k, \mathbf{Q}^k$
    $\mu^{k+1} \leftarrow$ using (6.28)
    $\mathbf{x}^{k+1} \leftarrow \mathbf{x}_r^k$ using (6.31)
    $k \leftarrow k + 1$
  until $(\mathcal{K} \leq \mathcal{K}_{tol}$ **or** $|f^k - f^{k-1}| \leq f_{tol})$ **and** $h \leq h_{tol}$
  $\mathbf{x}_r \leftarrow \mathbf{x}^*$
  $\mathbf{V}_r \leftarrow \mathbf{v}_r$ rank-one update using (6.12)
  $\mathbf{S}_{r,r} \leftarrow s_r$ rank-one update using (6.14)
  $\mathbf{W}_r \leftarrow \mathbf{w}_r$ rank-one update using (6.13)
  $r \leftarrow r + 1$
until $r = \sigma$
end

The vector-vector and matrix-vector primal-dual updates are of $\mathcal{O}(n)$, while the rank-one deflation update in (6.9), together with the function and gradient updates, are of $\mathcal{O}(pn)$ per mode r, requiring $\mathcal{O}(\sigma pn)$ operations for a rank-σ decomposition. Furthermore, all operations of $\mathcal{O}(pn)$ and $\mathcal{O}(n)$ are ideally suited for GPGPUs, which are memory bound on today's GPGPUs, as demonstrated in our numerical testing. Our approach, `saosvd-d`, is outlined in Algorithm 3.

6.6 An implicitly restarted scaling strategy

Our numerical testing has shown that without conditioning the convex problems $\mathcal{P}_{\text{PQ}}[k]$, convergence of an eigenvector may require many iterations. Additionally, generic solver termination conditions become difficult for singular values of differing magnitudes, since the KKT conditions are sensitive to the conditioning of $\mathcal{P}_{\text{PQ}}[k]$.

Hence, we propose a scaling strategy which relies on the magnitude of a preceding singular value, since the preceding singular value is either of equal magnitude, or larger, than the singular value being solved for. For a given singular-triplet r, we scale the objective function with a constant scalar, such that

$$\begin{aligned} f_r^k &\leftarrow f_r^k / s_{r-1}^2, \\ \boldsymbol{\nabla} f_r^k &\leftarrow \boldsymbol{\nabla} f_r^k / s_{r-1}^2. \end{aligned} \tag{6.32}$$

By definition, the singular values are ordered such that $s_r \leq s_{r-1}$ [23]. An upper bound of unity is therefore placed on the final function value if $s_r = s_{r-1}$, implying the presence of multiplicity. Since successive singular values do not differ by orders of magnitude in general,

this scaling strategy ensures that the convex subproblems are well-conditioned. However, if successive singular values do differ by more than an order of magnitude, we restart the optimization problem with a new scaling factor, by letting

$$f_r^k \leftarrow \frac{f_r^k}{|f_r(\mathbf{x}_r^+)|}, \qquad \boldsymbol{\nabla} f_r^k \leftarrow \frac{\boldsymbol{\nabla} f_r^k}{|f_r(\mathbf{x}_r^+)|}, \tag{6.33}$$

where $\mathbf{x}_r^+$ is the primal vector at initial convergence. This technique has served us well in our numerical testing, where singular values are prone to premature termination if the scaling factor used is too large. However, the premature termination is often within an order of magnitude of the exact value, allowing us to successfully use the proposed implicitly restarted method. For the first singular value, the scaling parameter is chosen as $s_0^2 = \mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x} \times 10^3$, where $\mathbf{x}$ is a uniformly distributed random vector in $(-1, 1)$.

6.7 Numerical experiments

We compare our naive[6] MATLAB implementation of `saosvd-d`, to MATLAB's built-in solver, `svds`, and the MATLAB implementation of the state-of-the-art Lanczos methods `irlba` [30] and PROPACKS's `lansvd` [106]. The interested reader is referred to [30] for information regarding `svds` and `irlba`. MATLAB R2019a, double precision computation and default settings are used for all the respective solvers, unless otherwise stated.

We use `saosvd-d-GPU` and `saosvd-s-GPU` to denote the GPGPU implementation of `saosvd-d` in double and single precision respectively, with `svds-GPU` denoting the GPGPU implementation of `svds`. The nature of our proposed method allows for a single precision implementation, in contrast to `lansvd`, `svds` and `svds-GPU`, which only support double precision computation. We reiterate that our algorithm is designed for GPGPU computation, with the serial implementation only presented as a general reference.

Since we strictly focus on in-core computation, matrix-vector operations are computationally cheap on today's GPGPUs, hence we accept that our method may require more matrix-vector operations compared to certain Lanczos methods. Of course, this could be prohibitively expensive for out-of-core computation, but the constant memory requirement of our method allows for difficult decompositions to be performed in-core, irrespective of the singular value distribution or number of desired singular-triplets.

For `saosvd-d-GPGPU` and `saosvd-s-GPGPU`, we use FORTRAN 90 with the cuBLAS library for all matrix-vector and vector-vector operations. Our test platform uses an 11 GB NVIDIA RTX 2080 Ti, an Intel Xeon-4112 CPU and 128 GB of 2666MHz system memory under Ubuntu 18.04. The drivers and libraries are NVIDIA 435, CUDA 10.1 and PGI 19.10; along with the pgf90 compiler and -O3 optimization flag invoked. Although it may not seem fair to compare a FORTRAN implementation to the MATLAB implementation of `svds-GPU`,

[6] No MATLAB mex functions were used for `saosvd-d`, whereas MATLAB's `svds` does indeed make use of pre-compiled mex functions.

`svds-GPU` and `svds` use C/FORTRAN pre-compiled mex files, hence the different routines are comparable.

In `saosvd-d`, each singular-triplet is initialized with a uniformly distributed random vector $\mathbf{x}_r^0 \in (-1, 1)$, before performing two iterations of the power-method, such that

$$\mathbf{x}_r^{k+1} \leftarrow \hat{\mathbf{A}}^{\mathrm{T}}\hat{\mathbf{A}}\mathbf{x}_r^k, \qquad \mathbf{x}_r^{k+2} \leftarrow \frac{\mathbf{x}_r^{k+1}}{\|\mathbf{x}_r^{k+1}\|_2}, \qquad k = 0, 1. \tag{6.34}$$

Using the power method for initialization has ensured local convergence across all our numerical tests, hence a mechanism for global convergence was not required. One iteration of the power method was used following final convergence of $\mathbf{x}_r^*$.

In Algorithm 3, we denote h as the constraint violation and $\mathcal{K}$ as the Euclidean norm of the KKT conditions. For all numerical testing, we used a constraint tolerance of $h_{\text{tol}} = 10^{-4}$, a KKT tolerance of $\mathcal{K}_{\text{tol}} = 10^{-3}$ and a relative function tolerance of $f_{\text{tol}} = 10^{-4}$. Although these tolerances do not seem overly strict, our scaling strategy ensures that accurate decompositions are still achieved, as demonstrated by our numerical results. N_f is used to denote the number of function and gradient evaluations required for convergence of a rank-σ decomposition, hence the total number of matrix-vector and vector-vector operations required is given by $2\,(N_f + 4\sigma - 1)$; noting that deflation is not required for the final singular-triplet.

To measure the accuracy of the decompositions, we use the Frobenius norm of the difference between $\mathbf{A}$ and the rank-σ approximation $\mathbf{A}_\sigma$, given in (6.2) by

$$\kappa = \|\mathbf{A} - \mathbf{A}_\sigma\|_F = \|\mathbf{A} - \mathbf{U}\mathbf{S}\mathbf{V}^{\mathrm{T}}\|_F. \tag{6.35}$$

The time measured, in seconds, for all algorithms throughout this study is purely the solver time, denoted by $t(s)$. Loading matrices from disk into memory, together with the time required to construct the test matrices, does not form part of the decomposition time. All test times are taken as the best of five runs for each problem, since all solvers use randomly distributed vectors at initialization.

An important note: All times for the GPGPU solvers, `saosvd-d-GPU`, `saosvd-s-GPU` and `svds-GPU` include the host-to-device transfer time of the test matrix $\mathbf{A}$, as well as the device-to-host transfer of $\mathbf{U}, \mathbf{S}$ and $\mathbf{V}$. This can contribute significantly to the overall computation time if the solver decomposition time is minimal. Our numerical testing indicated a consistent host-to-device transfer speed of approximately 7.8 GB/s.

Throughout all tables, **bold** text is used to denote the solver requiring the lowest decomposition time, provided the decomposition is correctly performed[7]. $t_{\text{CPU}}/t_{\text{GPU}}$ denotes the relative speedup between CPU and GPGPU implementations of a respective solver, while '—' is used to denote a failed decomposition. A summary of the test matrices used for the numerical testing can be found in Table 6.1.

[7]We exempt `saosvd-s-GPU` from GPGPU comparison, since comparing single to double precision computation is possibly unfair, despite `svds` not accepting single precision input matrices.

Table 6.1: The selected test problems. 'var' indicates that a respective parameter is subject to change, which we further indicate in the relevant table.

Matrix	p	n	Discipline	Reference
EXAMPLE 1	30000	var	Known singular values	Halko *et al.* [29]
EXAMPLE 2	30000	30000	Known singular values	Halko *et al.* [29]
MULT-DCOP-01	25187	25187	Circuit simulation problem	Florida collection [107]
BCSSTK25	15439	15439	Structural engineering	Boeing-Harwell [108]
MEMPLUS	17758	17758	Electronic circuit design	Boeing-Harwell [108]
FIDAP035	19716	19716	Finite element modeling	Boeing-Harwell [108]
FIDAPM11	22294	22294	Finite element modeling	Boeing-Harwell [108]
RAJAT10	30202	30202	Circuit simulation problem	Florida collection [107]
G7JAC120	35550	35550	Economic problem	Florida collection [107]
ONETONE1	36057	36057	Circuit simulation problem	Florida collection [107]
ONETONE2	36057	36057	Circuit simulation problem	Florida collection [107]

6.7.1 Example 1: multiplicity and slowly decaying singular values

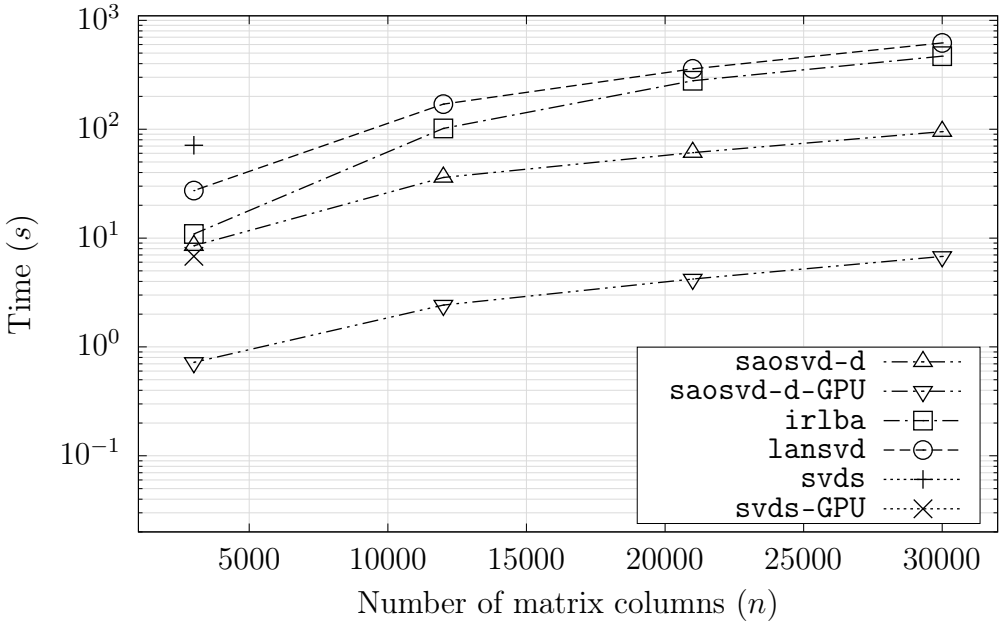


Figure 6.1: The decomposition times for the multiplicate and slowly-decaying singular values in Example 1, with varying column size n, fixed row size $p = 30000$ and fixed decomposition rank $\sigma = 12$.

To test solver performance on matrices containing multiplicity and slowly decaying singular values, we use a test matrix given by Halko *et al.* [29], whereby a matrix with known singular

values is constructed, such that

$$S_{j,j} = \begin{cases} 1.00, & j = 1, 2, \text{ or } 3 \\ 0.67, & j = 4, 5, \text{ or } 6 \\ 0.34, & j = 7, 8, \text{ or } 9 \\ 0.01, & j = 10, 11, \text{ or } 12 \\ 0.01 \times \frac{n-j}{n-13}, & j = 13, 14, \ldots, n \end{cases}$$

To construct a matrix with known singular values, we use the LAPACK routine `dlatms` [84]. Different test matrices with identical singular value distributions are constructed, by varying the column number n and keeping the row number $p = 30000$ constant, in a similar fashion to Halko *et al.* [29].

The CPU and GPGPU based solver results are presented in Tables 6.2 and 6.3 respectively, with Figure 6.1 providing a view of the change in time-complexity with n. The multiplicate and slowly decaying singular values make the problem particularly challenging, especially for the Lanczos methods, where `svds` can only successfully decompose $p = 3000$. `saosvd-d` clearly outperforms the Lanczos methods in this test, confirming the resilience of `saosvd-d` to multiplicity and slowly decaying singular values. With respect to `saosvd-d-GPU`, an order of magnitude reduction in time-complexity is observed compared to `saosvd-d`, demonstrating the ability of `saosvd-d` to exploit GPGPU performance.

6.7.2 Example 2: variably decaying singular values

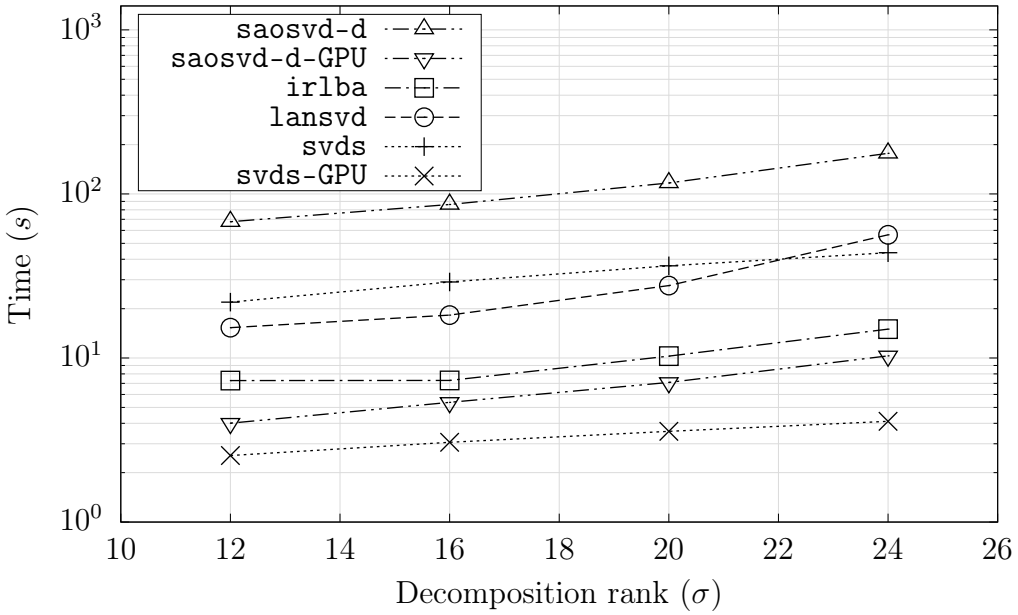


Figure 6.2: The decomposition times for the variably decaying singular values in Example 2, with varying decomposition rank-σ and fixed matrix size $p = n = 30000$.

To construct a matrix with well-separated singular values, which decay slowly after a predefined index, we again use the LAPACK routine `dlatms` [84] and follow a singular value distribution in [29], given by

$$S_{j,j} = \begin{cases} 10^{-4(j-1)/19}, & j = 1, 2, \ldots, 20, \\ 10^{-4}/(j-20)^{1/10}, & j = 21, 22, \ldots, n. \end{cases}$$

CPU and GPGPU numerical results are presented in Tables 6.4 and 6.5 respectively, with Figure 6.2 demonstrating the cost of solving for larger decomposition ranks σ. This example tests a solvers ability to find well-spaced singular values for $\sigma \leq 20$, after which the decomposition becomes significantly more challenging as the singular values decay slowly.

As expected, the subspace Lanczos methods are well-suited to this test, since minimal computational effort is required to solve for greater decomposition ranks that have well-separated singular values; a highly desirable property of methods acting on the entire subspace of desired vectors. `saosvd-d` solves for a single singular-triplet per optimization problem, hence the cost of performing a higher rank decomposition is relatively expensive for well-spaced singular values, especially when compared to subspace Lanczos methods. Notwithstanding the relatively poor performance of `saosvd-d`, `saosvd-d-GPU` still performs well compared to `svds-GPU`, again demonstrating the ability of `saosvd-d` to exploit GPGPU performance.

Despite being the fastest of the CPU Lanczos methods, `irlba` incorrectly performed the decomposition for $\sigma = 24$, highlighting the no-free-lunch (NFL) theorems of optimization [65].

6.7.3 Example 3: PCA of real-world datasets

To test the performance of the various algorithms on real-world datasets, we performed a principle component analysis (PCA) on matrices arising from various disciplines, given in Table 6.1. All test matrices were column mean centered and column unit normalized, ensuring fully dense test sets.

Tables 6.6 and 6.7 present the CPU and GPGPU numerical results respectively, where we perform low-rank $\sigma = 1, 5$ and 9 decompositions. The results are inline with expectations: for well-spaced singular values, `irlba` requires the lowest time-complexity, while `saosvd-d` performs best for slowly decaying singular values. Furthermore, when only the leading singular-triplet is desired, `saosvd-d` consistently outperforms the Lanczos methods, since the subspace advantages enjoyed by Lanczos methods are completely diminished for $\sigma = 1$.

Table 6.7 shows that `saosvd-s-GPU` and `saosvd-d-GPU` are both more robust and generally require lower time-complexity compared to `svds-GPU`. Notwithstanding that `svds` may require lower time-complexity compared to `saosvd-d` for certain tests, `saosvd-d-GPU` either outperforms or reduces the gap to `svds-GPU`; confirming the ability of `saosvd-d` to exploit GPGPU performance.

Finally, since `saosvd-d-GPU` involves only matrix-vector and vector-vector operations, we expect that `saosvd-d-GPU` is memory bound on today's GPGPUs. Indeed, our numerical results confirm this, with the single precision `saosvd-s-GPU` requiring approximately half the solver time compared to the double precision `saosvd-d-GPU`.

6.8 Conclusions and recommendations

Although a constrained convex optimization approach is nontraditional for the low-rank singular value decomposition (SVD) problem, we propose such a method herein. Compared to state-of-the-art Lanczos methods, our proposed method is relatively insensitive to multiplicity and slowly-decaying singular values, which is important for many practical problems of interest. Our proposed method requires constant memory and has the salient feature of embarrassingly parallel primal and dual variable updates, hence the method is ideal for implementation on massively parallel computational devices, such as general purpose graphical compute units (GPGPUs). A sequence of strictly convex, separable approximate quadratic-like optimization problems are constructed to solve for each singular-triplet, with each problem subject to a single constraint. Indeed, the approximate quadratic-like problems are efficiently solved for in the one-dimensional dual space, using a method inspired by the dual of Falk [70]. For the primal and dual updates to remain embarrassingly parallel, we solve for a single singular-triplet per optimization problem, with a Schur deflation technique used for further singular triplets.

We propose an implicitly restarted scaling strategy to ensure that the approximate convex functions are well-conditioned, which allows for generic solver settings across singular values of differing magnitudes, as well as minimal iterations required per singular-triplet. Our proposed scaling strategy relies on information from previously solved for singular values, thereby ensuring each optimization problem contains function values equal to or below unity at convergence.

Finally, we demonstrate both single and double precision GPGPU implementations of our method herein. Notwithstanding that the CPU implementation of our proposed method is robust and efficient compared to state-of-the-art Lanczos methods, the GPGPU implementations of our method convincingly outperform the GPGPU implementation of MATLAB's `svds` algorithm. We perform low-rank principle component analyses (PCA) on large-scale real-world matrices, and demonstrate that the CPU and GPGPU implementations of our proposed method are both efficient and robust.

6.9 Tables for numerical results

Table 6.2: Numerical results for the multiplicate and slowly-decaying singular values in Example 1, using CPU bound solvers, with varying column size n, fixed row size $p = 30000$ and fixed decomposition rank $\sigma = 12$.

		saosvd-d			irlba		lansvd		svds	
p	n	N_f	κ	$t\ (s)$	κ	$t\ (s)$	κ	$t\ (s)$	κ	$t\ (s)$
30000	30000	135	9.9981×10^{-1}	$\mathbf{9.48\times10^{1}}$	9.9981×10^{-1}	4.68×10^{2}	9.9981×10^{-1}	6.20×10^{2}	—	—
30000	21000	111	8.3643×10^{-1}	$\mathbf{6.10\times10^{1}}$	8.3643×10^{-1}	2.78×10^{2}	8.3643×10^{-1}	3.58×10^{2}	—	—
30000	12000	120	6.3215×10^{-1}	$\mathbf{3.61\times10^{1}}$	6.3215×10^{-1}	1.02×10^{2}	6.3215×10^{-1}	1.70×10^{2}	—	—
30000	3000	103	3.1562×10^{-1}	$\mathbf{8.51\times10^{0}}$	3.1562×10^{-1}	1.09×10^{1}	3.1562×10^{-1}	2.73×10^{1}	3.1562×10^{-1}	7.12×10^{1}

Table 6.3: Numerical results for the multiplicate and slowly-decaying singular values in Example 1, using GPGPU based solvers, with varying column size n, fixed row size $p = 30000$ and fixed decomposition rank $\sigma = 12$.

		saosvd-s-GPU			saosvd-d-GPU				svds-GPU		
p	n	N_f	κ	$t\ (s)$	N_f	κ	$t\ (s)$	$t_{\text{CPU}}/t_{\text{GPU}}$	κ	$t\ (s)$	$t_{\text{CPU}}/t_{\text{GPU}}$
30000	30000	123	9.9981×10^{-1}	2.57×10^{0}	141	9.9981×10^{-1}	$\mathbf{6.17\times10^{0}}$	15.36	—	—	—
30000	21000	120	8.3643×10^{-1}	1.76×10^{0}	127	8.3643×10^{-1}	$\mathbf{4.04\times10^{0}}$	15.08	—	—	—
30000	12000	124	6.3215×10^{-1}	1.04×10^{0}	152	6.3215×10^{-1}	$\mathbf{2.58\times10^{0}}$	13.96	—	—	—
30000	3000	125	3.1562×10^{-1}	2.70×10^{-1}	137	3.1562×10^{-1}	$\mathbf{6.45\times10^{-1}}$	13.19	3.1562×10^{-1}	6.80×10^{0}	10.47

Table 6.4: Numerical results for the variably decaying singular values in Example 2, using CPU bound solvers, with varying decomposition rank-σ and fixed matrix size $p = n = 30000$.

	saosvd-d			irlba		lansvd		svds	
σ	N_f	κ	$t\ (s)$	κ	$t\ (s)$	κ	$t\ (s)$	κ	$t\ (s)$
12	59	7.8704×10^{-3}	6.77×10^{1}	7.8704×10^{-3}	$\mathbf{7.29\times10^{0}}$	7.8704×10^{-3}	1.53×10^{1}	7.8704×10^{-3}	2.19×10^{1}
16	66	6.9258×10^{-3}	8.62×10^{1}	6.9258×10^{-3}	$\mathbf{7.29\times10^{0}}$	6.9258×10^{-3}	1.83×10^{1}	6.9258×10^{-3}	2.91×10^{1}
20	105	6.9049×10^{-3}	1.17×10^{2}	6.9049×10^{-3}	$\mathbf{1.03\times10^{1}}$	6.9049×10^{-3}	2.77×10^{1}	6.9049×10^{-3}	3.65×10^{1}
24	229	6.9024×10^{-3}	1.77×10^{2}	6.9026×10^{-3}	1.50×10^{1}	6.9024×10^{-3}	5.64×10^{1}	6.9024×10^{-3}	$\mathbf{4.38\times10^{1}}$

Table 6.5: Numerical results for the variably decaying singular values in Example 2, using GPGPU based solvers, with varying decomposition rank-σ and fixed matrix size $p = n = 30000$.

	saosvd-s-GPU			saosvd-d-GPU				svds-GPU		
σ	N_f	κ	$t\ (s)$	N_f	κ	$t\ (s)$	$t_{\text{CPU}}/t_{\text{GPU}}$	κ	$t\ (s)$c	
12	60	7.8704×10^{-3}	1.52×10^{0}	47	7.8704×10^{-3}	3.73×10^{0}	18.16	7.8704×10^{-3}	$\mathbf{2.54\times10^{0}}$	8.61
16	69	6.9258×10^{-3}	1.84×10^{0}	85	6.9258×10^{-3}	5.25×10^{0}	16.43	6.9258×10^{-3}	$\mathbf{3.06\times10^{0}}$	9.50
20	167	6.9049×10^{-3}	3.62×10^{0}	129	6.9049×10^{-3}	6.93×10^{0}	16.83	6.9049×10^{-3}	$\mathbf{3.58\times10^{0}}$	10.20
24	224	6.9024×10^{-3}	4.73×10^{0}	220	6.9024×10^{-3}	9.77×10^{0}	18.13	6.9024×10^{-3}	$\mathbf{4.11\times10^{0}}$	10.66

Table 6.6: Numerical results for the PCA performed in Example 3, using CPU bound solvers. — denotes failure of an algorithm to converge.

		saosvd-d			irlba		lansvd		svds	
Matrix	σ	N_f	κ	$t\ (s)$	κ	$t\ (s)$	κ	$t\ (s)$	κ	$t\ (s)$
MULT-DCOP-01	1	2	2.2676×10^{4}	$\mathbf{2.11\times10^{0}}$	2.2676×10^{4}	5.31×10^{0}	2.2676×10^{4}	6.93×10^{0}	2.2676×10^{4}	1.04×10^{1}
	5	116	2.2650×10^{4}	4.31×10^{1}	2.2650×10^{4}	$\mathbf{1.80\times10^{1}}$	2.2650×10^{4}	4.00×10^{1}	2.2650×10^{4}	1.07×10^{2}
	9	237	2.2630×10^{4}	8.63×10^{1}	2.2630×10^{4}	$\mathbf{1.80\times10^{1}}$	2.2630×10^{4}	3.69×10^{1}	2.2630×10^{4}	6.27×10^{1}
BCSSTK25	1	35	1.5435×10^{4}	$\mathbf{4.01\times10^{0}}$	1.5435×10^{4}	5.14×10^{0}	1.5435×10^{4}	1.52×10^{1}	1.5435×10^{4}	1.85×10^{1}
	5	172	1.5422×10^{4}	2.17×10^{1}	1.5422×10^{4}	$\mathbf{1.59\times10^{1}}$	1.5422×10^{4}	2.24×10^{1}	—	—
	9	302	1.5409×10^{4}	3.86×10^{1}	1.5409×10^{4}	$\mathbf{1.44\times10^{1}}$	1.5409×10^{4}	2.28×10^{1}	1.5409×10^{4}	1.18×10^{2}
MEMPLUS	1	6	1.7748×10^{4}	$\mathbf{1.53\times10^{0}}$	1.7748×10^{4}	2.57×10^{0}	1.7748×10^{4}	2.67×10^{1}	—	—
	5	28	1.7711×10^{4}	9.94×10^{0}	1.7711×10^{4}	$\mathbf{7.17\times10^{0}}$	1.7711×10^{4}	5.10×10^{1}	—	—
	9	51	1.7674×10^{4}	1.85×10^{1}	1.7674×10^{4}	$\mathbf{1.08\times10^{1}}$	1.7674×10^{4}	4.80×10^{1}	—	—
FIDAP035	1	25	1.9712×10^{4}	$\mathbf{5.04\times10^{0}}$	1.9712×10^{4}	1.34×10^{1}	1.9712×10^{4}	3.28×10^{1}	1.9712×10^{4}	5.86×10^{1}
	5	172	1.9698×10^{4}	3.50×10^{1}	1.9698×10^{4}	$\mathbf{2.05\times10^{1}}$	1.9698×10^{4}	4.97×10^{1}	1.9698×10^{4}	1.48×10^{2}
	9	291	1.9684×10^{4}	6.07×10^{1}	1.9684×10^{4}	$\mathbf{3.27\times10^{1}}$	1.9684×10^{4}	5.80×10^{1}	—	—
FIDAPM11	1	19	2.2286×10^{4}	$\mathbf{5.04\times10^{0}}$	2.2286×10^{4}	1.05×10^{1}	2.2286×10^{4}	3.06×10^{1}	2.2286×10^{4}	2.41×10^{1}
	5	151	2.2255×10^{4}	4.06×10^{1}	2.2255×10^{4}	$\mathbf{1.87\times10^{1}}$	2.2255×10^{4}	4.92×10^{1}	—	—
	9	255	2.2226×10^{4}	7.00×10^{1}	2.2226×10^{4}	$\mathbf{2.39\times10^{1}}$	2.2226×10^{4}	4.59×10^{1}	2.2226×10^{4}	5.88×10^{1}
RAJAT10	1	10	3.0191×10^{4}	$\mathbf{5.96\times10^{0}}$	3.0191×10^{4}	7.49×10^{0}	3.0191×10^{4}	9.83×10^{0}	3.0191×10^{4}	3.78×10^{2}
	5	149	3.0181×10^{4}	$\mathbf{7.37\times10^{1}}$	3.0181×10^{4}	2.10×10^{2}	3.0181×10^{4}	2.07×10^{2}	—	—
	9	267	3.0172×10^{4}	$\mathbf{1.34\times10^{2}}$	3.0172×10^{4}	2.97×10^{2}	3.0172×10^{4}	2.68×10^{2}	—	—
G7JAC120	1	12	3.5518×10^{4}	$\mathbf{9.17\times10^{0}}$	3.5518×10^{4}	1.82×10^{1}	3.5518×10^{4}	3.98×10^{1}	3.5518×10^{4}	4.50×10^{1}
	5	100	3.5400×10^{4}	7.60×10^{1}	3.5400×10^{4}	$\mathbf{2.23\times10^{1}}$	3.5400×10^{4}	5.29×10^{1}	3.5400×10^{4}	8.43×10^{1}
	9	202	3.5297×10^{4}	1.51×10^{2}	3.5297×10^{4}	$\mathbf{3.65\times10^{1}}$	3.5297×10^{4}	7.43×10^{1}	3.5297×10^{4}	1.20×10^{2}
ONETONE1	1	23	3.6054×10^{4}	$\mathbf{1.50\times10^{1}}$	3.6054×10^{4}	7.69×10^{1}	3.6054×10^{4}	1.48×10^{2}	3.6054×10^{4}	5.26×10^{2}
	5	144	3.6046×10^{4}	$\mathbf{1.01\times10^{2}}$	3.6046×10^{4}	7.23×10^{2}	3.6046×10^{4}	3.47×10^{2}	—	—
	9	262	3.6038×10^{4}	$\mathbf{1.87\times10^{2}}$	3.6038×10^{4}	7.34×10^{2}	3.6038×10^{4}	3.04×10^{2}	—	—
ONETONE2	1	35	3.6055×10^{4}	$\mathbf{2.13\times10^{1}}$	3.6054×10^{4}	1.02×10^{2}	3.6054×10^{4}	2.03×10^{2}	3.6054×10^{4}	5.26×10^{2}
	5	175	3.6047×10^{4}	$\mathbf{1.18\times10^{2}}$	3.6047×10^{4}	2.29×10^{2}	3.6047×10^{4}	2.86×10^{2}	—	—
	9	311	3.6039×10^{4}	$\mathbf{2.13\times10^{2}}$	3.6039×10^{4}	2.26×10^{2}	3.6039×10^{4}	2.54×10^{2}	3.6039×10^{4}	7.68×10^{2}

Table 6.7: Numerical results for the PCA performed in Example 3, using GPGPU based solvers. $t_{\text{CPU}}/t_{\text{GPU}}$ indicates speedup achieved compared to the respective CPU implementation of a solver, while — denotes failure of an algorithm to converge.

		saosvd-s-GPU			saosvd-d-GPU				svds-GPU		
Matrix	σ	N_f	κ	$t\ (s)$	N_f	κ	$t\ (s)$	$t_{\text{CPU}}/t_{\text{GPU}}$	κ	$t\ (s)$	$t_{\text{CPU}}/t_{\text{GPU}}$
MULT-DCOP-01	1	2	2.2677×10^{4}	7.12×10^{-2}	2	2.2677×10^{4}	$\mathbf{7.78\times10^{-1}}$	2.72	2.2676×10^{4}	1.55×10^{0}	6.68
	5	97	2.2651×10^{4}	1.42×10^{0}	78	2.2650×10^{4}	$\mathbf{2.79\times10^{0}}$	15.45	2.2650×10^{4}	9.15×10^{0}	11.69
	9	226	2.2630×10^{4}	3.19×10^{0}	224	2.2630×10^{4}	6.16×10^{0}	14.01	2.2630×10^{4}	$\mathbf{5.63\times10^{0}}$	11.14
BCSSTK25	1	32	1.5436×10^{4}	1.72×10^{-1}	37	1.5435×10^{4}	$\mathbf{5.57\times10^{-1}}$	7.21	1.5435×10^{4}	1.92×10^{0}	9.66
	5	167	1.5422×10^{4}	8.71×10^{-1}	181	1.5422×10^{4}	$\mathbf{1.81\times10^{0}}$	11.97	—	—	—
	9	322	1.5409×10^{4}	1.67×10^{0}	304	1.5409×10^{4}	$\mathbf{2.92\times10^{0}}$	13.22	1.5409×10^{4}	1.03×10^{1}	11.52
MEMPLUS	1	6	1.7749×10^{4}	6.03×10^{-2}	6	1.7748×10^{4}	$\mathbf{4.22\times10^{-1}}$	3.63	—	—	—
	5	33	1.7712×10^{4}	2.96×10^{-1}	32	1.7711×10^{4}	$\mathbf{8.91\times10^{-1}}$	11.15	—	—	—
	9	60	1.7675×10^{4}	5.32×10^{-1}	61	1.7674×10^{4}	$\mathbf{1.40\times10^{0}}$	13.17	—	—	—
FIDAP035	1	23	1.9711×10^{4}	2.04×10^{-1}	28	1.9712×10^{4}	$\mathbf{7.98\times10^{-1}}$	6.32	1.9712×10^{4}	5.15×10^{0}	11.37
	5	170	1.9697×10^{4}	1.41×10^{0}	168	1.9698×10^{4}	$\mathbf{2.69\times10^{0}}$	13.00	1.9698×10^{4}	1.22×10^{1}	12.08
	9	290	1.9684×10^{4}	2.41×10^{0}	310	1.9684×10^{4}	$\mathbf{4.63\times10^{0}}$	13.11	—	—	—
FIDAPM11	1	24	2.2287×10^{4}	2.73×10^{-1}	24	2.2286×10^{4}	$\mathbf{9.68\times10^{-1}}$	5.21	2.2286×10^{4}	2.53×10^{0}	9.53
	5	143	2.2256×10^{4}	1.55×10^{0}	128	2.2255×10^{4}	$\mathbf{2.91\times10^{0}}$	13.96	—	—	—
	9	279	2.2227×10^{4}	3.00×10^{0}	312	2.2226×10^{4}	6.14×10^{0}	11.41	2.2226×10^{4}	$\mathbf{5.27\times10^{0}}$	11.16
RAJAT10	1	5	3.0193×10^{4}	1.57×10^{-1}	5	3.0191×10^{4}	$\mathbf{1.19\times10^{0}}$	5.03	3.0191×10^{4}	3.03×10^{1}	12.45
	5	150	3.0184×10^{4}	2.99×10^{0}	121	3.0182×10^{4}	$\mathbf{5.15\times10^{0}}$	14.31	—	—	—
	9	265	3.0172×10^{4}	5.28×10^{0}	274	3.0172×10^{4}	$\mathbf{1.01\times10^{1}}$	13.27	—	—	—
G7JAC120	1	18	3.5515×10^{4}	5.46×10^{-1}	19	3.5518×10^{4}	$\mathbf{2.28\times10^{0}}$	4.02	3.5518×10^{4}	4.96×10^{0}	9.08
	5	94	3.5397×10^{4}	2.73×10^{0}	107	3.5400×10^{4}	$\mathbf{6.63\times10^{0}}$	11.47	3.5400×10^{4}	8.09×10^{0}	10.41
	9	221	3.5294×10^{4}	6.20×10^{0}	189	3.5297×10^{4}	$\mathbf{1.08\times10^{1}}$	13.89	3.5297×10^{4}	1.09×10^{1}	11.01
ONETONE1	1	27	3.6054×10^{4}	8.01×10^{-1}	28	3.6055×10^{4}	$\mathbf{2.67\times10^{0}}$	5.64	3.6054×10^{4}	4.34×10^{1}	12.11
	5	147	3.6046×10^{4}	4.22×10^{0}	144	3.6046×10^{4}	$\mathbf{8.29\times10^{0}}$	12.20	—	—	—
	9	288	3.6038×10^{4}	8.18×10^{0}	278	3.6038×10^{4}	$\mathbf{1.48\times10^{1}}$	12.67	—	—	—
ONETONE2	1	27	3.6054×10^{4}	8.01×10^{-1}	37	3.6055×10^{4}	$\mathbf{3.03\times10^{0}}$	7.02	3.6054×10^{4}	4.34×10^{1}	12.11
	5	175	3.6046×10^{4}	4.94×10^{0}	185	3.6047×10^{4}	$\mathbf{1.00\times10^{1}}$	11.77	—	—	—
	9	348	3.6038×10^{4}	9.75×10^{0}	359	3.6039×10^{4}	$\mathbf{1.81\times10^{1}}$	11.75	3.6039×10^{4}	6.26×10^{1}	12.27

Chapter 7

A low-rank singular value decomposition algorithm

The work presented here originates from a paper titled "A sequential approximate optimization approach for low-rank singular value decomposition" [109], which is under review at the time of writing. The paper is co-authored by Prof. Albert A. Groenwold.

7.1 Abstract

We propose a sequential approximate optimization approach for low-rank singular value decomposition, by solving a sequence of orthogonally constrained approximate quadratic-like problems that maximize the Rayleigh quotient. The approximate quadratic-like subproblems are solved for using a feasible descent method, with the salient feature of embarrassingly parallel, closed-form expressions available for both the primal and dual variable updates. Both primal and dual variable updates are ideal for implementation on massively parallel computational devices, such as general purpose graphical compute units (GPGPUs), which we demonstrate herein. Since optimization based methods are sensitive to scaling, we propose an implicitly restarted scaling strategy for the objective function, which relies upon preceding sequential singular value information.

Our method has a few salient features; namely *a priori* known constant memory requirements, resilience to multiplicate and slowly decaying singular values and covariance-free operations; the latter being crucial for sparse matrix decompositions. We compare our method to state-of-the-art Lanczos methods for selected test sparse matrices, and demonstrate the efficacy of our method both by means of competitive decomposition times, as well as accuracy. Our method is the only method to correctly perform all decompositions in this study, demonstrating the robustness of our approach compared to state-of-the-art Lanczos methods. Furthermore, we show numerical results for a GPGPU implementation of our method, which we compare to a GPGPU implementation of MATLAB's Lanczos `svds` algorithm. We demonstrate that our method is both more robust, as well as able to leverage parallel computation to a far greater degree than the GPGPU implementation of `svds`.

7.2 Introduction

Consider an equality constrained nonlinear problem $\mathcal{P}_{\text{NLP}}$ of the form

$$\begin{aligned} \min_{\mathbf{x}} \ & f(\mathbf{x}) \\ \text{subject to} \ & g_j(\mathbf{x}) = 0, \qquad j = 1, \ldots, m, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, \qquad i = 1, \ldots, n, \end{aligned} \tag{7.1}$$

where $f(\mathbf{x}) \in \mathcal{R}$ is an objective function and the $g_j(\mathbf{x})$, $j = 1, 2, \cdots, m$ are m equality constraint functions, which are dependent on the n primal variables $\mathbf{x} = \{x_1, x_2, \cdots, x_n\}^{\mathrm{T}} \in \mathcal{X} \subset \mathcal{R}^n$. $\check{x}_i$ and $\hat{x}_i$ are the lower and upper bounds on variable x_i respectively, such that $\mathcal{X}$ is bounded and closed.

Sequential approximate optimization (SAO), which is arguably the state-of-the-art in structural optimization, relies upon the iterative solution of a sequence of approximate optimization problems $\mathcal{P}_{\text{P}}[k]$, $k = 0, 1, 2, \ldots$; where $\mathcal{P}_{\text{P}}[k]$ is constructed from simple approximate functions, such that at some iterate $\mathbf{x}^k$

$$\begin{aligned} \min_{\mathbf{x}} \ & \tilde{f}^k(\mathbf{x}) \\ \text{subject to} \ & \tilde{g}_j^k(\mathbf{x}) = 0, \qquad j = 1, \ldots, m, \\ & \check{x}_i^k \leq x_i \leq \hat{x}_i^k, \qquad i = 1, \ldots, n, \end{aligned} \tag{7.2}$$

the approximate primal continuous problem $\mathcal{P}_{\text{P}}[k]$ is formed. (7.2) contains n unknown primal variables, m equality constraints and $2n$ bound constraints requiring minimal additional computational effort, as we will show in sections to come.

Notably in SAO, the approximate subproblem $\mathcal{P}_{\text{P}}[k]$ is both *separable* and *convex*; a consequence of the prohibitively expensive storage and evaluation of second-order information in structural optimization. Hence, reliance is placed on intervening variables as Hessian terms, which when substituted into a Taylor series expansion, hopefully reveal the behavior prevalent in the original functions they approximate. The approximate functions then become linear in the intervening variables used, allowing for simple closed-form updates of the intervening variables at each iteration $\mathbf{x}^k$.

Without loss of generality, we consider a real matrix $\mathbf{A}$, which we assume is either square, or has more rows p than columns n (i.e. $p \geq n$)[1]. $\mathbf{A}$ can be written in its closest approximate rank-σ form by means of the Eckart-Young-Mirsky theorem [25], as

$$\mathbf{A}_\sigma = \mathbf{W}\mathbf{S}\mathbf{V}^{\mathrm{T}}, \tag{7.3}$$

where $\mathbf{A}_\sigma$ is

$$\begin{aligned} \min_{\mathbf{A}_\sigma} \ & \|\mathbf{A} - \mathbf{A}_\sigma\|_F \\ \text{subject to} \ & \mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{I}, \end{aligned} \tag{7.4}$$

[1] If $n > p$, we use $\mathbf{A}^{\mathrm{T}}$ instead.

and $\|\cdot\|_F$ denotes the Frobenius norm. $\mathbf{W}$ is a $p \times \sigma$ orthonormal matrix containing the left-singular vectors of $\mathbf{A}$, $\mathbf{S}$ is a $\sigma \times \sigma$ square diagonal matrix with the descending singular values, and $\mathbf{V}$ is an $n \times \sigma$ orthonormal matrix containing the right-singular vectors of $\mathbf{A}$ [23, 25].

In this study, we focus on computing a low-rank singular value decomposition (SVD), such that $\sigma \ll \min(p, n)$, since for big-data applications it is often not feasible, nor desired, that the full-rank decomposition be computed. The full-rank decomposition is computationally expensive, with the best known algorithm requiring $\mathcal{O}(4p^2n + 22n^3)$ floating point operations (flops) [23], thereby being prohibitively expensive for large problem sizes. The low-rank SVD is ubiquitous with applications in information retrieval [99, 110, 111], linear least squares, engineering [99] and principal component analysis (PCA) [29], to name but a few.

The exploitation of Krylov subspaces, on which recently popularized low-rank decomposition methods are based, use some variation of the Arnoldi or Lanczos bidiagonalization method [29, 30]. Although these methods prove extremely fruitful in the eigenvalue and eigenvector extraction of positive semi-definite matrices, they do suffer from shortcomings inherent to any method exploiting the Krylov subspace. They often require many iterations for challenging problems, thereby requiring large memory and computational demands [98], which can make in-core computation on massively parallel computational devices, such as general purpose graphical compute units (GPGPUs), challenging. Additionally, they suffer from numerical instability due to propagated round-off errors and struggle to converge if the singular values of $\mathbf{A}$ contain multiplicity or decay slowly [29, 30], a property found in many practical applications [96, 97]. Techniques to ameliorate these issues include preconditioning and restarting; but as effective as they have become, do not completely resolve the underlying issues inherent to methods exploiting the Krylov subspace.

Although possibly not obvious at first, the problem in (7.3), like many structural optimization problems, contains prohibitively expensive second-order information; both in evaluation and storage. This makes the use of intervening variables, and therefore a SAO approach, attractive for solving (7.4). As previously mentioned, we are focused herein on low-rank approximations of $\mathbf{A}$, which importantly results in far fewer constraints than primal variables ($m \ll n$). Together, these two properties of (7.3) warrant the investigation of using a *dual* SAO method, with dual methods proving highly effective for problems where $n/m \gg 1$, since $\mathcal{R}^m$ is a significantly lower dimensional space than $\mathcal{R}^n$. To the authors knowledge, there is no literature regarding a dual SAO method for low-rank singular value decomposition, which is somewhat surprising given that the problem in (7.4) is ideally posed for a dual method, from a theoretical standpoint at least[2]. Importantly, we further demonstrate that using a dual approach results in *closed-form*, *embarrassingly parallel* updates for *both* the primal and dual variables, allowing for a straightforward GPGPU implementation of our proposed method.

Despite the inherent nature of the problem being attractive for a dual method, we further investigate a mathematical optimization approach in the hope of overcoming the shortcomings prevalent in state-of-the-art Krylov methods. Our method has *fixed, a priori known* memory requirements, and is resilient in the presence of multiplicity or slowly decaying singular

[2]For most practical problems of interest, there are far more primal variables than constraints for the low-rank decomposition problem.

values. Furthermore, the time-complexity of CPU bound Lanczos methods dramatically increases with problem dimensionality, and since GPGPU implementations of the method are more challenging because of the inherent nature of Lanczos methods[3], literature on their development is scarce. In contrast, our method contains barely any serial bottlenecks of significant time-complexity, and can exploit GPGPU performance to a far greater degree than the Krylov methods compared with herein, as we will show in sections to come.

Like all methods, our solver performance increases with well-separated singular values. However, convergence is still achieved even in cases of multiplicity. We provide numerical examples to demonstrate this, for both artificial and real-world large sparse datasets, compared to state-of-the-art CPU Lanczos methods. We implement a MATLAB CPU bound variation of our algorithm, denoted by `saosvd`, which provides a fair numerical testing ground to the MATLAB CPU bound Lanczos methods we compare with herein. Furthermore, since we are interested in both the CPU and GPGPU implementation of our algorithm, we implement a variation in a FORTRAN CUDA GPGPU context, denoted by `saosvd-GPU`, which we compare to the GPGPU accelerated implementation of MATLAB's `svds`.

The contributions of this study are summarized as follows: We introduce a formal definition for the constrained Rayleigh maximization problem in Section 7.3, followed by the quadratic-like approximations we make use of throughout in Section 7.4. Next, we present a solver that has embarrassingly parallel, closed-form updates for both the primal and dual variables in Section 7.5, followed by a unique scaling approach for the objective function in Section 7.6. In Section 7.7, we introduce a sparse pseudo-deflation power method, which is both a useful starting vector strategy, as well as a means for final convergence. Finally, we present numerical results in Section 7.8 for the decomposition of selected test sparse matrices, before making conclusions and recommendations in Section 7.9.

7.3 Problem formulation

It is well-known that the right-singular vectors correspond to the eigenvectors of the covariance structure $\mathbf{A}^{\mathrm{T}}\mathbf{A}$. Departing with the well-known eigenvector and eigenvalue relationship [26]

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x}_r = \lambda_r \mathbf{x}_r, \tag{7.5}$$

and left-multiplying by $\mathbf{x}_r^{\mathrm{T}}$, we obtain

$$\mathbf{x}_r^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x}_r = \mathbf{x}_r^{\mathrm{T}}\lambda_r \mathbf{x}_r. \tag{7.6}$$

We note that $\mathbf{x}_r^{\mathrm{T}}\mathbf{x}_r = \|\mathbf{x}_r\|_2^2$ and λ_r are scalars and solve for the eigenvalue of the covariance structure

$$\lambda_r = \frac{\mathbf{x}_r^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x}_r}{\|\mathbf{x}_r\|_2^2}, \tag{7.7}$$

which is the so-called Rayleigh quotient [27, 28].

[3] One reason being that they may not have fixed, *a priori known* memory requirements.

To obtain the eigenvalue associated with the most variance, the expression in (7.7) needs to be maximized, which we do by *minimizing* the constrained problem

$$\begin{aligned} \min_{\mathbf{x}_r} \; f(\mathbf{x}_r) &= -\frac{\mathbf{x}_r^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x}_r}{\|\mathbf{x}_r\|_2^2} \qquad r = 2, \ldots, \sigma \\ g_j(\mathbf{x}_r) &= \mathbf{x}_r^{\mathrm{T}}\mathbf{v}_j = 0 \qquad j = 1, \ldots, r-1 = m, \end{aligned} \tag{7.8}$$

where f and g_j represent the objective and constraint functions respectively. The vector $\mathbf{v}_j$ represents a respective column from the matrix $\mathbf{V}$ containing previously solved for right-singular vectors. In (7.8), the constraints ensure that a given right-singular vector is orthogonal to all previously solved for right-singular vectors, as per the definition in (7.4). Hence, $r = 1, \ldots, \sigma$ problems of the form (7.8) are solved for a rank-σ decomposition, where subscript r denotes the index of the desired right-singular vector. Although the first mode, $r = 1$, can be solved for in an unconstrained fashion, our numerical testing has shown it to be computationally expensive, since $\mathcal{R}^n$ may be a highly dimensional search space. Instead, we slightly reformulate (7.8), ensuring that our dual method can be exploited, such that

$$\begin{aligned} \min_{\mathbf{x}_r} \; f(\mathbf{x}_r) &= -\mathbf{x}_r^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x}_r \qquad r = 1 \\ g_r(\mathbf{x}_r) &= \mathbf{x}_r^{\mathrm{T}}\mathbf{x}_r - 1 = 0 \qquad r = m, \end{aligned} \tag{7.9}$$

which explicitly constrains the feasible search space to the unit sphere $\in \mathcal{R}^n$. The problem in (7.9) contains almost identical properties to the problem in (7.8), hence we reduce repetition by only referring to (7.8), unless otherwise stated.

The problem in (7.8) is convex, since the stationary points of (7.8) are unique to the desired eigenvector $\mathbf{x}_r^*$ when (7.8) is minimized [5, 27, 33]. Although it is possible to solve for all right-singular vectors in a single optimization problem, our numerical testing has shown it to be inefficient, since the number of primal variables grows by $n \times \sigma$. Instead, we solve the problem in an *incremental* fashion, by solving for a sequence of right-singular vectors $\mathbf{v}_1, \ldots, \mathbf{v}_\sigma$. The incremental approach allows us to solve for a single n-dimensional primal vector σ-times, significantly reducing the required solution time-complexity. This is especially important for large sparse matrices, where memory requirements alone may prohibit a vector of size $n \times \sigma$.

The corresponding eigenvector of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ represents the right-singular vector, $\mathbf{v}_r$, of $\mathbf{A}$; which in terms of the solution vector is

$$\mathbf{v}_r = \frac{\mathbf{x}_r^*}{\|\mathbf{x}_r^*\|_2}. \tag{7.10}$$

The left-singular vectors, $\mathbf{w}_r$, are then analytically determined using [99]

$$\mathbf{w}_r = \frac{\mathbf{A}\mathbf{v}_r}{s_r}, \tag{7.11}$$

where the scalar

$$s_r = \sqrt{-f(\mathbf{x}_r^*)} = \sqrt{\lambda_r}, \tag{7.12}$$

is the respective singular value [105]. The eigenvalues of the symmetric, positive semi-definite matrix in (7.8) are positive and real, ensuring that $s_r \in \mathcal{R}$.

We choose to solve for the right-singular vectors from our definition of $p \geq n$, since it results in a smaller $n \times n$ covariance matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$. Alternatively, should, $p \leq n$, the covariance structure $\mathbf{A}\mathbf{A}^{\mathrm{T}}$ may be used in (7.8) to solve for the left-singular vectors. Furthermore, the singular values may be solved for in ascending order by maximizing (7.8), but this may be difficult or impossible due to numerical instabilities, especially since we implicitly form $\mathbf{A}^{\mathrm{T}}\mathbf{A}$.

An important note: we never explicitly form the covariance structures in our function or gradient updates. Despite $\mathbf{A}$ being sparse, $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ may be significantly less sparse; losing both the computational and memory advantages associated with sparsity. Thus, we note that

$$\mathbf{x}_r^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x}_r = (\mathbf{A}\mathbf{x}_r)^{\mathrm{T}}(\mathbf{A}\mathbf{x}_r),$$

hence updating (7.8) requires a single matrix-vector operation, followed by the inner-product of the resulting vector, together with the inner-product of the primal vector.

7.4 Quadratic-like approximations

To solve for (7.8) in a SAO fashion, quadratic-like approximations are used to construct convex approximate subproblems. For the sake of brevity, we let $\mathbf{x}_r = \mathbf{x}$ throughout the remainder of this study. For each right-singular vector $\mathbf{x}_r$, $r = 1, \ldots, \sigma$ problems (7.8) are solved with an updated set of $m = r - 1$ constraints. Since only the number of constraints changes per problem (7.8), we denote the primal variable (right-singular vector) for a given problem r as $\mathbf{x}$. The approximate functions are constructed in exactly the same fashion for each mode r, with the only difference being an increase in constraints as r is incremented after each $\mathbf{x}_r^*$ is solved for.

We now introduce a simplified notation for zero- and first-order information, where

$$\begin{aligned} f^k &= f(\mathbf{x}^k), \\ \left(\frac{\partial f}{\partial x_i}\right)^k &= \frac{\partial f(\mathbf{x}^k)}{\partial x_i}, \\ g_j^k &= g_j(\mathbf{x}^k), \\ \left(\frac{\partial g_j}{\partial x_i}\right)^k &= \frac{\partial g_j(\mathbf{x}^k)}{\partial x_i}, \end{aligned}$$

and

$$\begin{aligned}
\boldsymbol{\nabla} f^k &= \left[\left(\frac{\partial f}{\partial x_1}\right)^k, \ldots, \left(\frac{\partial f}{\partial x_n}\right)^k\right]^{\mathrm{T}}, \\
\boldsymbol{\nabla} g_j^k &= \left[\left(\frac{\partial g_j}{\partial x_1}\right)^k, \ldots, \left(\frac{\partial g_j}{\partial x_n}\right)^k\right]^{\mathrm{T}}, \\
\mathbf{0}_m &= [0, \ldots, 0]^{\mathrm{T}} \in \mathcal{R}^m, \\
\mathbf{g}^k &= \left[g_1^k, \ldots, g_m^k\right]^{\mathrm{T}}, \\
\boldsymbol{\nabla} \mathbf{g}^k &= \begin{bmatrix} — & \boldsymbol{\nabla} g_1^k & — \\ — & \vdots & — \\ — & \boldsymbol{\nabla} g_m^k & — \end{bmatrix} = \begin{bmatrix} — & \mathbf{v}_1 & — \\ — & \vdots & — \\ — & \mathbf{v}_{r-1} & — \end{bmatrix} \in \mathcal{R}^{m \times n}.
\end{aligned}$$

We reiterate that for all modes, excluding the first where $m = 1$, $m = r - 1$ to ensure orthogonality between the right-singular vectors.

The approximate subproblems constructed are based on an incomplete series expansion (ISE) [2], with approximations $\tilde{f}^k(\mathbf{x})$ and $\tilde{\mathbf{g}}^k(\mathbf{x})$ constructed at the point $\mathbf{x}^k$ to the objective and constraint functions respectively, such that

$$\begin{aligned}
\tilde{f}^k(\mathbf{x}) &= f^k + \boldsymbol{\nabla} f^{k\mathrm{T}} \mathbf{s} + \frac{1}{2} \mathbf{s}^{\mathrm{T}} \mathbf{C}^k \mathbf{s}, \\
\tilde{\mathbf{g}}^k(\mathbf{x}) &= \mathbf{g}^k + \boldsymbol{\nabla} \mathbf{g}^k \mathbf{s},
\end{aligned} \tag{7.13}$$

with $\mathbf{s} = (\mathbf{x} - \mathbf{x}^k)$ and $\mathbf{C}^k$ an appropriate approximate *diagonal* Hessian matrix. For all modes except the first, which we address further in this section, the constraints are linear and thus contain no Hessian information. Hence, for all modes excluding the first, the approximate Lagrangian Hessian is equivalent to the Hessian of the approximate objective function.

We will herein consider a simple instance of the approximate Hessian $\mathbf{C}^k$, in which the curvature is chosen such that the approximate function value at the previous iterate, $\tilde{f}^{k-1}$, is equal to the real function value, f^{k-1}, at the previous iterate; being a spherical quadratic approximation (denoted SPH-QDR). To construct a spherical quadratic approximation [54], we select $c^k \equiv c_i^k \ \forall \ i$, which requires the determination of the single unknown c^k, obtained by enforcing the condition

$$\tilde{f}(\mathbf{x}^{k-1}) = f(\mathbf{x}^{k-1}), \tag{7.14}$$

which implies that

$$c^k = \frac{2[f(\mathbf{x}^{k-1}) - f(\mathbf{x}^k) - (\boldsymbol{\nabla} f(\mathbf{x}^k))^{\mathrm{T}}(\mathbf{x}^{k-1} - \mathbf{x}^k)]}{\|\mathbf{x}^{k-1} - \mathbf{x}^k\|_2^2}. \tag{7.15}$$

The approximate Hessian is then formed by letting $\mathbf{C}^k = c^k \mathbf{I}$, resulting in the approximation proposed by Snyman and Hay [54], with an alternative condition for formulating a spherical quadratic approximation presented in Reference [55]. In the first iteration when $k = 0$, a curvature $c^0 = 10$ is assumed, since no historic information is available.

The approximations (7.13) are (diagonal) quadratic, allowing primal problem $\mathcal{P}_{\text{NLP}}$ to be trivially transformed into a sequence of convex sequential quadratic programs (SQP) $\mathcal{P}_{\text{PQ}}[k]$, written as

□ *Quadratic approximate program* $\mathcal{P}_{\text{PQ}}[k]$

$$\begin{aligned} \min_{\mathbf{s}} \ & \tilde{f}^k(\mathbf{s}) = f^k + \boldsymbol{\nabla} f^{k\mathrm{T}}\mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}\mathbf{Q}^k\mathbf{s} \\ \text{subject to} \ & \tilde{\mathbf{g}}^k(\mathbf{s}) = \mathbf{g}^k + \boldsymbol{\nabla}\mathbf{g}^k\mathbf{s} = \mathbf{0}_m \\ & \check{x}_i^k \leq x_i \leq \hat{x}_i^k, \qquad\qquad i = 1, 2, \ldots, n, \end{aligned} \tag{7.16}$$

with $\mathbf{Q}^k$ the Hessian matrix of the approximate Lagrangian $\tilde{\mathcal{L}}^k$. For details, we refer the reader to Etman *et al.* [52, 53]. Since the Lagrangian multipliers $\boldsymbol{\mu}^{k*}$ at the solution of subproblem $\mathcal{P}_{\text{PQ}}[k]$ are unknown, the multipliers $\boldsymbol{\mu}^k$ are used to construct the quadratic program. As previously mentioned, we note that the constraints for $r > 1$ are linear, thereby resulting in the approximate Lagrangian Hessian

$$\begin{aligned} q^k &= \begin{cases} \max\left(\epsilon,\ c^k + 2\mu_1^k\right), & r = 1, \\ \max\left(\epsilon,\ c^k\right), & r > 1, \end{cases} \\ \mathbf{Q}^k &= q^k\mathbf{I}. \end{aligned} \tag{7.17}$$

In the case of $r = 1$, the Hessian formed by the constraint $g_1(\mathbf{x}) = \mathbf{x}^{\mathrm{T}}\mathbf{x} - 1$ is a constant $2\mathbf{I}$, hence the Hessian of the Lagrangian is given by $(c^k + 2\mu_1^k)\mathbf{I}$.

To enforce convexity, we require the approximate Lagrangian Hessian to be (semi) positive definite. Since $\mathbf{Q}^k$ is diagonal, positive definiteness simply requires the individual diagonal elements Q_{ii}^k to be positive. From (7.17), this requires that ϵ be a "small" positive scalar, which we rather arbitrarily selected as 10^{-6} for all our numerical tests.

7.5 A closed-form SAO method

Since dual SAO optimization methods today are so well-known, we will mainly outline a variant used herein. Examples of established SAO algorithms include the method of moving asymptotes (MMA) of Svanberg [8], and the CONLIN algorithm of Fleury and Braibant [10], and many derivatives thereof.

The convergence of SAO to local optima for the approximate subproblems $\mathcal{P}_{\text{PQ}}[k]$, under the assumption that a mechanism is used to enforce global convergence, is well established; with one such example found in [7]. As previously mentioned, the $\mathcal{P}_{\text{PQ}}[k]$ are simple *separable* quadratic-like convex functions, with many algorithms able to find efficient solutions for $\mathcal{P}_{\text{PQ}}[k]$, by exploiting the separability therein. One such algorithm is the dual of Falk [70], which is able to exploit the underlying separability of $\mathcal{P}_{\text{PQ}}[k]$, allowing for the efficient solution of large-scale problems subject to few constraints. Further work on the dual of Falk, presented in Chapter 5, has resulted in a feasible descent method[4] with closed-form solutions for the primal and dual variables, although only the primal variables are updated

[4]For equality constrained problems such as (7.8).

in an embarrassingly parallel fashion. In Chapter 5, we specifically demonstrated the efficacy of the algorithm for large-scale problems subject to few constraints, which is exactly the case for (7.8). For general problems, the algorithm in Chapter 5 requires that the dual variables be solved for as a solution to a linear system. However, the first-order conditions of (7.8) possess a special structure, which results in embarrassingly parallel updates for *both* the dual and primal variables.

We depart by closely following the method in Chapter 5, using the well-known Lagrangian for constrained optimization, given by

$$\tilde{\mathcal{L}}^k(\mathbf{x}, \boldsymbol{\mu}) = \tilde{f}^k(\mathbf{x}) + \tilde{\mathbf{g}}^k(\mathbf{x})^{\mathrm{T}}\boldsymbol{\mu}. \tag{7.18}$$

The separable and convex approximate functions allow the Falk dual [70] to be invoked, resulting in the so-called dual function, where

$$\begin{aligned} \tilde{\gamma}(\boldsymbol{\mu}) = \quad & \min_{\mathbf{x}} \tilde{\mathcal{L}}^k(\mathbf{x}, \boldsymbol{\mu}), \\ \text{subject to} \quad & \mathbf{x} \in \mathcal{X}. \end{aligned} \tag{7.19}$$

Since only equality constraints are present in $\mathcal{P}_{\mathrm{PQ}}[k]$, the dual variables in (7.19) are unrestricted in sign. A salient feature of the Falk dual [70], under the assumption that the approximate functions are separable and convex and that $\mathcal{X}$ is bounded and closed, is that the primal minimizer is available as a closed-form expression $\mathbf{x} = \mathbf{x}(\boldsymbol{\mu})$. Note that (7.19) will attain a minimum for all $\boldsymbol{\mu} \in \mathcal{R}^m$, although a respective minimum may not be unique, hence the primal minimizer of $\min_{\mathbf{x}} \tilde{\mathcal{L}}^k(\mathbf{x}, \boldsymbol{\mu})$ is a function of $\boldsymbol{\mu}$. Thus, with the slight abuse of notation, the dual function of (7.19) may be given as [83]

$$\begin{aligned} \tilde{\gamma}(\boldsymbol{\mu}) = \quad & \min_{\mathbf{x}} \tilde{\mathcal{L}}^k(\mathbf{x}(\boldsymbol{\mu}), \boldsymbol{\mu}), \\ \text{subject to} \quad & \mathbf{x}(\boldsymbol{\mu}) \in \mathcal{X}. \end{aligned} \tag{7.20}$$

Assuming that the rows of the constraint Jacobian $\boldsymbol{\nabla}\mathbf{g}(\mathbf{x}^*)$ are linearly independent, which holds for the orthogonal eigenvectors of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$, then unique $\boldsymbol{\mu}^* \in \mathcal{R}^m$ exists such that

$$\boldsymbol{\nabla} f(\mathbf{x}^*) + \boldsymbol{\nabla}\mathbf{g}(\mathbf{x}^*)^{\mathrm{T}}\boldsymbol{\mu}^* = 0, \tag{7.21}$$

which is the well-known KKT conditions. At the desired eigenvector $\mathbf{x}^*$, the KKT conditions imply that $\mathbf{x}^*$ is a stationary point of (7.18). To determine the dual solution vector $\boldsymbol{\mu}^*$, we maximize the dual function (7.19)

$$\tilde{\gamma}^k(\boldsymbol{\mu}^*) = \max_{\boldsymbol{\mu}} \tilde{\mathcal{L}}^k(\mathbf{x}(\boldsymbol{\mu}), \boldsymbol{\mu}), \quad \boldsymbol{\mu} \in \mathcal{R}^m. \tag{7.22}$$

Note that we have dropped the requirement that $\mathbf{x}(\boldsymbol{\mu}) \in \mathcal{X}$. We can ensure that $\mathbf{x}(\boldsymbol{\mu}^*)$ is an interior-point of $\mathcal{X}$ by choosing suitable bounds on the primal variables and employing a simple projection strategy, thus enabling the embarrassingly parallel primal and dual variable updates. In essence, we use the algorithm presented in Chapter 5 without the primal-projection strategy that handles solutions residing on the bounds of $\mathcal{X}$, since we can guarantee that $\mathbf{x}(\boldsymbol{\mu}^*)$ is an interior-point of $\mathcal{X}$.

Traditionally, (7.22) is maximized using iterative solvers, since although primal separability exists, which allows for closed-form primal updates, the dual variables are coupled if second-order constraint information is required. However, in Chapter 5 we demonstrated that a closed-form expression exists which maximizes (7.22), provided that a SQP formulation is used to construct (7.22), as opposed to the popular pure dual formulation often used in structural optimization. By enforcing primal first-order optimality conditions, we can derive the closed-form primal updates initially shown by Falk [70], such that

$$\boldsymbol{\nabla}_{\mathbf{x}}\tilde{\mathcal{L}}^k(\mathbf{x},\boldsymbol{\mu}) = \boldsymbol{\nabla} f^k + \mathbf{Q}^k(\mathbf{x}-\mathbf{x}^k) + \boldsymbol{\nabla}\mathbf{g}^{k\mathrm{T}}\boldsymbol{\mu} = 0, \tag{7.23}$$

resulting in the closed-form primal update

$$\mathbf{x}(\boldsymbol{\mu}) = \mathbf{x}^k - \mathbf{E}^k\left(\boldsymbol{\nabla} f^k + \boldsymbol{\nabla}\mathbf{g}^{k\mathrm{T}}\boldsymbol{\mu}\right), \tag{7.24}$$

where $\mathbf{E}^k = (\mathbf{Q}^k)^{-1}$. For numerical stability, we bound the primal variables

$$\mathbf{x}(\boldsymbol{\mu}) = \Pi\left(\mathbf{x}^k - \mathbf{E}^k\left(\boldsymbol{\nabla} f^k + \boldsymbol{\nabla}\mathbf{g}^{k\mathrm{T}}\boldsymbol{\mu}\right)\right), \tag{7.25}$$

where $\Pi(\cdot)$ is a projection of variable x_i on $[\check{x}_i^k, \hat{x}_i^k]$, ensuring a closed and bound primal set $\mathcal{X} \in \mathcal{R}^n$. The update in (7.25), including the projection $\Pi(\cdot)$, can be performed in an embarrassingly parallel fashion.

For ease of notation and brevity in what is to follow, let

$$\mathbf{s}(\boldsymbol{\mu}) = \mathbf{x}(\boldsymbol{\mu}) - \mathbf{x}^k = -\mathbf{E}^k\left(\boldsymbol{\nabla} f^k + \boldsymbol{\nabla}\mathbf{g}^{k\mathrm{T}}\boldsymbol{\mu}\right), \tag{7.26}$$

which can be once continuously differentiated as

$$\boldsymbol{\nabla}_{\boldsymbol{\mu}}\mathbf{s}(\boldsymbol{\mu}) = -\mathbf{E}^k\boldsymbol{\nabla}\mathbf{g}^{k\mathrm{T}}. \tag{7.27}$$

Next, we expand (7.22) with the zero-, first- and second-order conditions used to create the approximate functions, such that

$$\max_{\boldsymbol{\mu}}\ \tilde{\gamma}^k(\boldsymbol{\mu}) = f^k + \boldsymbol{\nabla} f^{k\mathrm{T}}\mathbf{s}(\boldsymbol{\mu}) + \frac{1}{2}\mathbf{s}(\boldsymbol{\mu})^{\mathrm{T}}\mathbf{Q}^k\mathbf{s}(\boldsymbol{\mu}) + \boldsymbol{\mu}^{\mathrm{T}}\left(\mathbf{g}^k + \boldsymbol{\nabla}\mathbf{g}^k\mathbf{s}(\boldsymbol{\mu})\right), \tag{7.28}$$

and solve for the first-order optimality conditions, now with respect to $\boldsymbol{\mu}$

$$\boldsymbol{\nabla}_{\boldsymbol{\mu}}\tilde{\gamma}^k(\boldsymbol{\mu}) = -\boldsymbol{\nabla}\mathbf{g}^k\mathbf{E}^k\boldsymbol{\nabla} f^k + \mathbf{g}^k - \boldsymbol{\nabla}\mathbf{g}^k\mathbf{E}^k\boldsymbol{\nabla}\mathbf{g}^{k\mathrm{T}}\boldsymbol{\mu}^* = 0. \tag{7.29}$$

This results in a "dual" linear system[5] almost identical to that found in Chapter 5, where

$$\boldsymbol{\nabla}\mathbf{g}^k\mathbf{E}^k\boldsymbol{\nabla}\mathbf{g}^{k\mathrm{T}}\boldsymbol{\mu}^* = \mathbf{g}^k - \boldsymbol{\nabla}\mathbf{g}^k\mathbf{E}^k\boldsymbol{\nabla} f^k. \tag{7.30}$$

By noting that we use the spherical quadratic Hessian approximation $\mathbf{Q}^k = q^k\mathbf{I}$ in Section 7.4, $\mathbf{E}^k$ results in the simple scaling

$$\boldsymbol{\nabla}\mathbf{g}^k\boldsymbol{\nabla}\mathbf{g}^{k\mathrm{T}}\boldsymbol{\mu}^* = q^k\mathbf{g}^k - \boldsymbol{\nabla}\mathbf{g}^k\boldsymbol{\nabla} f^k. \tag{7.31}$$

[5]The first-order conditions given in (7.30) may be obtained by evaluating the constraint functions at $\tilde{\mathbf{g}}^k(\mathbf{x}(\boldsymbol{\mu}^*)) = 0$; a known result from literature.

Since we require that all previously solved for eigenvectors are orthogonal,

$$\boldsymbol{\nabla}\mathbf{g}^k\boldsymbol{\nabla}\mathbf{g}^{k\mathrm{T}} = \mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{I},$$

which leads to the embarrassingly parallel, closed-form dual update

$$\boldsymbol{\mu}^* = q^k\mathbf{g}^k - \boldsymbol{\nabla}\mathbf{g}^k\boldsymbol{\nabla}f^k. \tag{7.32}$$

For the first mode, $r = 1$, the dual update remains closed-form, since we require the determination of a single dual variable, given by

$$\mu_1^* = \left(q^k g^k - \boldsymbol{\nabla}g^k\boldsymbol{\nabla}f^k\right)/\|\boldsymbol{\nabla}g_1^k\|_2^2. \tag{7.33}$$

Finally, we bound the dual variables following the update in (7.32), enforcing

$$\boldsymbol{\mu}^* = \Pi\left(q^k\mathbf{g}^k - \boldsymbol{\nabla}\mathbf{g}^k\boldsymbol{\nabla}f^k\right), \tag{7.34}$$

where $\Pi(\cdot)$ is a projection of dual variable μ_i on $[\check{\mu}_i^k, \hat{\mu}_i]$ and $\hat{\mu}_j = -\check{\mu}_j$ some large value, say 10^8. Bounding the dual variables, under the assumption that the problem is unimodal, guarantees the convergence of the SAO sequence to a feasible point [6], allowing us to circumvent the requirement for explicit relaxation variables. This has benefits both in terms of computational effort, as well as reduced memory requirements; a technique that has been successfully used for the dual of Falk in [6].

The embarrassingly parallel primal (7.25) and dual (7.34) updates are ideally suited for GPGPU implementation, a result we demonstrate in sections to come. Both the dual and primal updates require a cost of $\mathcal{O}(nr)$ for each mode solved, resulting from matrix-vector operations, which are ideally suited for parallel computational devices. For large problem sizes, this can result in a significant reduction in the required time-complexity of the problem, with initial results suggesting our method is able to exploit GPGPU performance to a greater degree than Lanczos methods[6].

Lastly, we make use of the filtered conservatism in [1], which combines the strategies of conservatism and a trust region with a nonlinear acceptance filter, and in doing so enforces global convergence. We use identical parameters to those given in [1], with the exception being an inner iteration limit of $l \leq 50$. Additionally, we update the first-order conditions after updating the function values required by filtered-conservatism, since doing so is relatively inexpensive, especially on GPGPUs. Our method, `saosvd`, is outlined in Algorithm 4.

7.6 Implicit restarting for objective function scaling

Scaling the objective function is critical to the performance of Algorithm 4, since singular values of orders of magnitude difference may be solved for in a single optimization problem. Selecting appropriate termination criteria can be made difficult, or even impossible, for such cases, because the gradients of the objective function in (7.8) are strongly coupled to the

[6] See for example Figure 7.4.

Algorithm 4: Algorithm `saosvd` for rank-σ SVD

Initialize: $r = 1$, $\mathbf{A} \in \mathcal{R}^{p\times n}$, $\mathbf{W} \in \mathcal{R}^{p\times\sigma}$, $\mathbf{S} \in \mathcal{R}^{\sigma\times\sigma}$, $\mathbf{V} \in \mathcal{R}^{n\times\sigma}$
repeat
  Initialize: Problem (7.8) at current index r, $k = 0$, $\mathbf{x}^0 \in \mathcal{R}^n$, $\boldsymbol{\mu}^0 \in \mathcal{R}^m$
  repeat
    Update $f^k, \boldsymbol{\nabla} f^k, \mathbf{g}^k, \boldsymbol{\nabla}\mathbf{g}^k, \mathbf{Q}^k$
    $\boldsymbol{\mu}^{k+1} \leftarrow$ using (7.34)
    $\mathbf{x}^{k+1} \leftarrow$ using (7.25)
    Accept/reject iterate $\mathbf{x}^{k+1}$ using filtered conservatism
    $k \leftarrow k + 1$
  until ($\mathcal{K} \leq \mathcal{K}_{tol}$ **or** $|f^k - f^{k-1}| \leq f_{tol}$) **and** $h \leq h_{tol}$
  $\mathbf{V}_r \leftarrow \mathbf{v}_r$ using (7.10)
  $\mathbf{S}_{r,r} \leftarrow s_r$ using (7.12)
  $\mathbf{W}_r \leftarrow \mathbf{w}_r$ using (7.11)
  $r \leftarrow r + 1$
until $r = \sigma$
end

magnitude of the objective function value. Possibly surprising at first, this is easy to verify, with (7.8) clearly bounded above by zero and below by $-s_r^2$. Hence, larger singular values form more contoured surfaces on which we minimize, with the surfaces formed by smaller singular values being much 'flatter'. The flatness of certain surfaces can cause undesirable effects, such as the appearance of local minima [34], which we have observed in the presence of slowly decaying singular values together with relaxed convergence criteria, despite the eigenvectors being relatively accurate.

To palliate this issue, we propose a scaling strategy based on the magnitude of the preceding singular value

$$\begin{aligned} f(\mathbf{x}_r) &\leftarrow f(\mathbf{x}_r)/s_{r-1}^2 \qquad r = 2, \ldots, \sigma, \\ \boldsymbol{\nabla}_{\mathbf{x}_r} f(\mathbf{x}_r) &\leftarrow \boldsymbol{\nabla}_{\mathbf{x}_r} f(\mathbf{x}_r)/s_{r-1}^2. \end{aligned} \tag{7.35}$$

From the definition of the singular values in (7.12) that $s_r \leq s_{r-1}$, an upper bound of unity is placed on the final function value if $s_r = s_{r-1}$, implying the presence of multiplicity. Since successive singular values do not differ by orders of magnitude in general (with a somewhat worst-case example being the Marchenko-Pastur distribution [112]), this scaling strategy ensures that problem convergence is not falsely terminated for small singular values $s_r \ll 1$, or made too challenging for large singular values $s \gg 1$.

However, should there be greater than an order of magnitude difference between successive singular values, we propose using an implicitly restarted strategy, such that

$$f(\mathbf{x}_r) \leftarrow -\frac{f(\mathbf{x}_r)}{f(\mathbf{x}_r^+)} \qquad r = 2, \ldots, \sigma,$$

where $\mathbf{x}_r^+$ is the primal variable at initial convergence, i.e. $s_{r-1}^2/|f(\mathbf{x}_r^+)| \geq 10$. Our numerical testing has shown that $f(\mathbf{x}_r^+)$ is often near to $f(\mathbf{x}_r^*)$, resulting in few additional iterations

to converge to the correct function value $f(\mathbf{x}_r^*)$. Nevertheless, this scaling strategy has proved fruitful in our mathematical optimization approach, hence we imply the use of it from hereon. For the first mode, we select $s_0^2 = \mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x} \times 10^3$, where $\mathbf{x}$ is a uniformly randomly distributed vector $\in (-1, 1)$.

7.7 A sparse pseudo-deflation power method

Algorithm 4 is highly sensitive to the initial choice of starting vector, with problem convergence heavily dependent on an appropriate strategy. Hence, we propose the use of a Schur complement deflation technique on $\mathbf{A}$ [104], in conjunction with the power method, to drive the initial vector towards the desired singular value. Schur deflation removes the variance associated with previously found eigenvectors [104], allowing the power method to drive the initial vector towards the maximum of the remaining variance.

Since $\mathbf{V}_{r-1}$ is orthonormal, the Schur deflation technique at a given singular value index r, without explicitly forming the covariance structure, is given by

$$\mathbf{A} \leftarrow \mathbf{A} - \mathbf{A}\mathbf{V}_{r-1}\mathbf{V}_{r-1}^{\mathrm{T}} \tag{7.36}$$

where $\mathbf{V}_{r-1}$ is the orthonormal matrix containing the previously found $r-1$ right-singular vectors.

The proposed pseudo-deflation power method is then

$$\begin{aligned} \mathbf{x}_r^{k+1} &\leftarrow \left(\mathbf{A} - \mathbf{A}\mathbf{V}_{r-1}\mathbf{V}_{r-1}^{\mathrm{T}}\right)^{\mathrm{T}} \left(\mathbf{A} - \mathbf{A}\mathbf{V}_{r-1}\mathbf{V}_{r-1}^{\mathrm{T}}\right) \mathbf{x}_r^k, \\ \mathbf{x}_r^{k+1} &\leftarrow \mathbf{x}_r^{k+1} / \|\mathbf{x}_r^{k+1}\|_2. \end{aligned} \tag{7.37}$$

It is important to note that (7.37) can be performed using solely matrix-vector operations, thus requiring minimal computational and memory requirements. The use of this technique has greatly enhanced the performance of our algorithm, where we choose a uniformly distributed initial vector $\mathbf{x}_r^0 \in (-1, 1)$, before using two iterations of (7.37). Furthermore, following convergence of (7.8), we perform a final two iterations of (7.37) on $\mathbf{x}_r^*$ to achieve greater levels of accuracy.

7.8 Numerical experiments

We specifically focus on the decomposition of sparse matrices in this study, since they arise in many practical applications of interest. For a sparse matrix $\mathbf{A}$, we shall denote the cardinality of the number of nonzero elements as

$$\zeta = \mathrm{card}(\mathbf{A}), \tag{7.38}$$

which leads to the sparsity fraction

$$\gamma = \frac{\zeta}{p \times n}. \tag{7.39}$$

A matrix is considered sparse if $\zeta \ll p \times n$, resulting in the ratio $\gamma \ll 1$.

We compare our MATLAB implementation of `saosvd` to MATLAB's built-in solver, `svds`, and the MATLAB implementation of the state-of-the-art Lanczos methods `irlba`[7] [30] and `lansvd` [106]; the latter a routine in the robust PROPACK library. The MATLAB solvers are able to exploit the associated memory and computational advantages of sparse matrix structures, by applying the `sparse` routine on a dense matrix $\mathbf{A}$. Default settings are used for all solvers, unless otherwise stated.

For `saosvd-GPU`, we use CUDA FORTRAN 90 with the cuBLAS and cuSPARSE libraries for all matrix-vector and vector-vector operations. We note that cuBLAS is known to perform poorly on 'tall-skinny' matrices, such as $\mathbf{V}$, which has far more rows than columns, but we make use of the appropriate cuBLAS library regardless. Our test platform includes a NVIDIA RTX 2080 Ti 11 GB, an Intel Xeon-4112 CPU and 128 GB of 2666 MHz system memory under Ubuntu 18.04. The drivers and libraries are MATLAB R2019a, NVIDIA 418, CUDA 10.1 and PGI 19.04; along with the pgf90 compiler and invoking the -O3 optimization flag. All FORTRAN sparse matrices are stored in compressed sparse row (CSR) format, allowing for an easy interface to the cuSPARSE library. To optimize memory access patterns and reduce the overall decomposition time, we store both $\mathbf{A}$ as well as $\mathbf{A}^{\mathrm{T}}$ in GPGPU memory. The computation and storage of $\mathbf{A}^{\mathrm{T}}$ from $\mathbf{A}$ is performed on the GPGPU, and included in the reported overall decomposition time. All methods make use of double precision computation, with decomposition times reported as the lowest over five runs, since all methods make use of random initial vectors. To ensure that $\mathcal{X}$ is bounded and closed when using the projection $\Pi(\cdot)$, we enforce

$$\hat{x}_i^k = -\check{x}_i^k = \begin{cases} 10^0, & r = 1, \\ 10^7, & r > 1, \end{cases} \quad , \quad \forall\ k, \quad i = 1, \ldots, n.$$

We now introduce the absolute maximum constraint violation h, the relative function step size $f_{\mathrm{rel}} = |f(\mathbf{x}^k) - f(\mathbf{x}^{k-1})|$ and the Euclidean norm of the KKT conditions, $\mathcal{K}$. Problem execution is terminated when $h \leq 10^{-10}$ ($h \leq 10^{-4}$ when of $r = 1$) and either $\mathcal{K} \leq 10^{-5}$, or $f_{\mathrm{rel}} \leq 10^{-5}$ ($f_{\mathrm{rel}} \leq 10^{-4}$ when $r = 1$), is met. We choose convergence tolerances of moderate accuracy to ensure that further modes are not penalized by accumulated error, since we reiterate that subsequent modes are solved for in an incremental fashion. A tight tolerance is placed on the constraint violation h, preventing loss of orthogonality between the desired eigenvectors.

Our starting vectors are initialized such that $\mathbf{x}^0 \in (-1, 1)$ and $\boldsymbol{\mu}^0 \in (-1, 1)$ are uniformly distributed, followed by two iterations of the sparse pseudo-deflation technique (7.37) in Section 7.7. Lastly, two iterations of sparse pseudo-deflation (7.37) are applied to the solution vector $\mathbf{x}_r^*$ to achieve even greater levels of accuracy.

All problem execution times are measured in seconds and exclude the construction of the test matrices, focusing solely on the low-rank decomposition time-complexity. Furthermore, our times include all host-to-device[8] and device-to-host[9] data transfers for the GPGPU-based

[7] The interested reader is referred to [30] for information regarding `svds` and `irlba`.

[8] The host-to-device transfers include moving $\mathbf{A}$ into GPGPU memory.

[9] The device-to-host transfers include moving $\mathbf{U}$, $\mathbf{S}$ and $\mathbf{V}$ into host memory from GPGPU memory.

solvers, negating an unfair advantage by pre-loading data onto the GPGPU. For problems of low time-complexity, the data transfer costs may form a significant fraction of the total time.

Determining the accuracy of the decomposition for large sparse matrices is non-trivial. Since the low-rank SVD produces a dense $p \times n$ resultant matrix, it can require prohibitively expensive memory to calculate the Frobenius norm in (7.4). To determine the accuracy of decompositions throughout this study, we use the method of Halko *et al.* [29]. Given a residual matrix

$$\mathbf{D} = \mathbf{A} - \mathbf{WSV}^{\mathrm{T}}, \tag{7.40}$$

the spectral norm of the residual matrix, using the power method, can be estimated by

$$p_{j,\sigma}(\mathbf{D}) = \max \sqrt{\frac{\|(\mathbf{D}^{\mathrm{T}}\mathbf{D})^{j}\boldsymbol{\omega}^{q}\|_2}{\|(\mathbf{D}^{\mathrm{T}}\mathbf{D})^{j-1}\boldsymbol{\omega}^{q}\|_2}}, \quad q = 1, \ldots, \sigma. \tag{7.41}$$

Clearly, (7.41) is bounded above by the exact spectral norm $\|\mathbf{D}\|_2 = \delta_0$ [29]. An involved analysis in [29] shows with *high* probability[10] that $p_{j,\sigma}(\mathbf{D})$ is bounded below by

$$p_{j,\sigma}(\mathbf{D}) \geq \delta_0/2. \tag{7.42}$$

Thus, with high probability a decomposition was correctly performed if

$$\delta_0/2 \leq p_{j,\sigma}(\mathbf{D}) \leq \delta_0(1 + \epsilon_D), \tag{7.43}$$

where we select $\epsilon_D = 10^{-3}$ as some small value[11], with $j = 6$ and σ as the rank of the decomposition performed, as in [29]. The n-length vectors $\boldsymbol{\omega}^q$ are uniform randomly distributed, with a mean of zero and unit variance.

We compare the approximate spectral norm, $\delta = p_{j,\sigma}(\mathbf{D})$, in (7.41) to the known best spectral norm, δ_0, by computing the ratio

$$\kappa = \frac{\delta - \delta_0}{\delta_0}, \tag{7.44}$$

such that a decomposition is correctly performed if

$$-0.5 \leq \kappa \leq \epsilon_D. \tag{7.45}$$

However, a probabilistic analysis in [31] (again mentioned in [29]) shows that the *expected* error lies within the range

$$-0.1 \lesssim \kappa \lesssim \epsilon_D, \tag{7.46}$$

which we indeed observe throughout our numerical testing. For matrices where δ_0 is unknown, we make use of the robust `lansvd` solver to calculate the value[12]. Our numerical

[10] Across all our numerical testing, the lower probability bound was 99.9994%.

[11] For well spaced singular values, $p_{j,\sigma}(\mathbf{D})$ may compute an almost exact $\|\mathbf{D}\|_2$, hence we do not wish to penalize methods that are within a small tolerance above $\|\mathbf{D}\|_2$.

[12] An exception is the NLPKKT160 test matrix, where we used `irlba`, since `lansvd` failed to perform the decomposition.

testing confirms the robustness of this approach, since the majority of solvers converge to almost identical values of δ_0 solved for by `lansvd`.

The details of the test matrices used in our numerical testing are given in Table 7.1, with the solvers requiring the lowest time-complexity, provided the decomposition was correctly performed, highlighted in bold in all further tables. It is important to note that we do not exploit any symmetry in our test matrices, since doing so would provide a substantial advantage.

Table 7.1: The selected test problems. 'var' indicates that a respective parameter is subject to change, which we further indicate in the relevant table.

Matrix	p	n	γ	Discipline	Reference
EXAMPLE 1	100000	100000	1.01×10^{-3}	Known singular values	Halko *et al.* [29]
EXAMPLE 2	100000	var	var	Known singular values	Halko *et al.* [29]
EXAMPLE 3	100000	100000	var	Random matrices	
MEMPLUS	17758	17758	3.14×10^{-4}	Electronic circuit design	Boeing-Harwell [108]
AF23560	23560	23560	8.30×10^{-4}	Computational fluid dynamics	Boeing-Harwell [108]
S3DKQ4M2	90449	90449	5.41×10^{-4}	Structural mechanics	Boeing-Harwell [108]
S3DKT3M2	90449	90449	4.51×10^{-4}	Structural mechanics	Boeing-Harwell [108]
PRE2	659033	659033	1.34×10^{-5}	Electronic circuit design	Florida collection [107]
NLPKKT160	8345600	8345600	3.24×10^{-6}	Nonlinear PDE optimization	Florida collection [107]

7.8.1 Example 1: variably decaying singular values

In this example, we construct a matrix with known singular values, which initially are well-spaced, followed by singular values that decay slowly. We follow the singular value distribution in [29], by constructing a test matrix with the singular values

$$S_{j,j} = \begin{cases} 10^{-4(j-1)/19}, & j = 1, 2, \ldots, 20, \\ 10^{-4}/(j-20)^{1/10}, & j = 21, 22, \ldots, n, \end{cases}$$

where the singular values are well spaced for $\sigma < 20$, after which they decay slowly. To ensure the matrix is sufficiently sparse, we construct asymmetric band matrices by exploiting the LAPACK routine `dlatms` [84]. Band matrices, with known singular values, of bandwidth B are constructed, with the only nonzero entries contained in the diagonals of height B above and below the main diagonal. Rather arbitrarily, we select $B = 50$; ensuring that the matrix is sufficiently sparse for the values of p and n tested. As given in Table 7.1, we select $n = p = 1 \times 10^5$, resulting in a sparsity fraction of $\gamma = 1.01 \times 10^{-3}$, and again follow [29] by investigating the performance of the solvers for differing decomposition ranks.

The decomposition times, presented in Figure 7.1, and accuracy are presented in Tables 7.2 and 7.3, for the CPU and GPGPU bound methods respectively. This is a somewhat worst case scenario for `saosvd`, since Lanczos methods have high convergence rates for well-spaced singular values. `irlba`, which *failed to perform the decomposition correctly* for $\sigma = 24$,

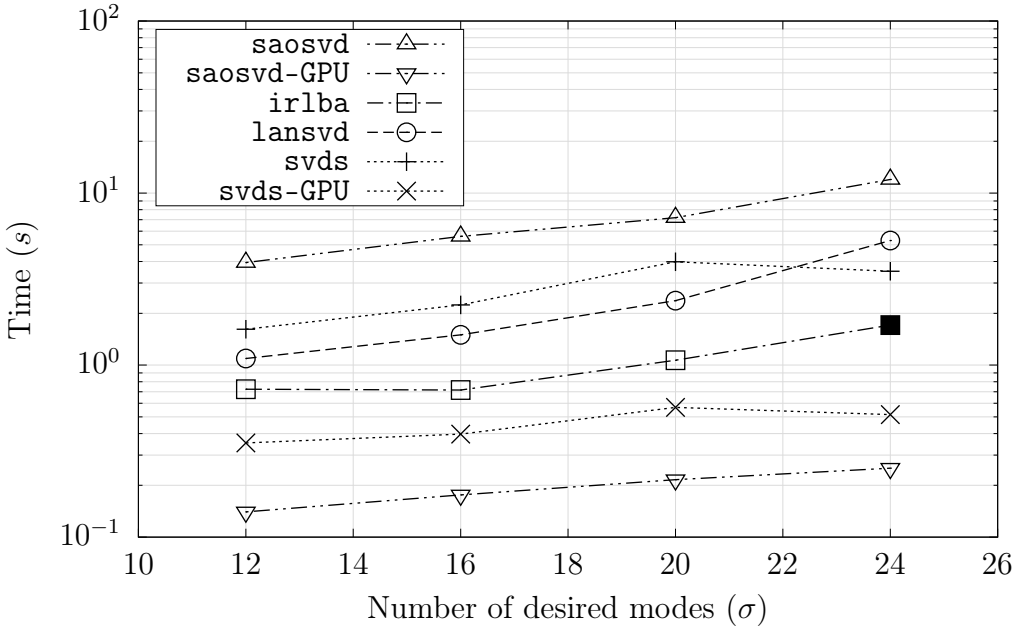


Figure 7.1: The decomposition times required for the variably decaying singular values in Example 1, with $p = n = 1 \times 10^5$ and varying decomposition rank-σ. ■ indicates that the decomposition was performed incorrectly, despite the algorithm converging.

required the lowest-time complexity of the CPU bound solvers, with `saosvd` requiring the highest time-complexity, albeit not by orders of magnitude.

Of the GPGPU-accelerated methods, `saosvd-GPU` outperforms `svds-GPU`, despite `svds` outperforming `saosvd`. `svds-GPU`, which is a Krylov method, is not able to exploit GPGPU performance to the same degree as `saosvd-GPU`; a theme that we observe throughout our numerical testing.

7.8.2 Example 2: multiplicity and slowly decaying singular values

This example focuses on singular values that contain multiplicity and decay slowly, a problem known to be challenging for methods exploiting Krylov subspaces. To do so, we make use of another singular value distribution presented in [29], where

$$S_{j,j} = \begin{cases} 1.00, & j = 1, 2, \text{ or } 3 \\ 0.67, & j = 4, 5, \text{ or } 6 \\ 0.34, & j = 7, 8, \text{ or } 9 \\ 0.01, & j = 10, 11, \text{ or } 12 \\ 0.01 \times \frac{n-j}{n-13}, & j = 13, 14, \ldots, n, \end{cases}$$

again exploiting the LAPACK routine `dlatms`. The singular values contain multiplicity for $\sigma \leq 12$, after which their values decay slowly. Band matrices with $B = 50$ are constructed, ensuring that the matrix is sufficiently sparse for our test purposes. We select $p = 1 \times 10^5$

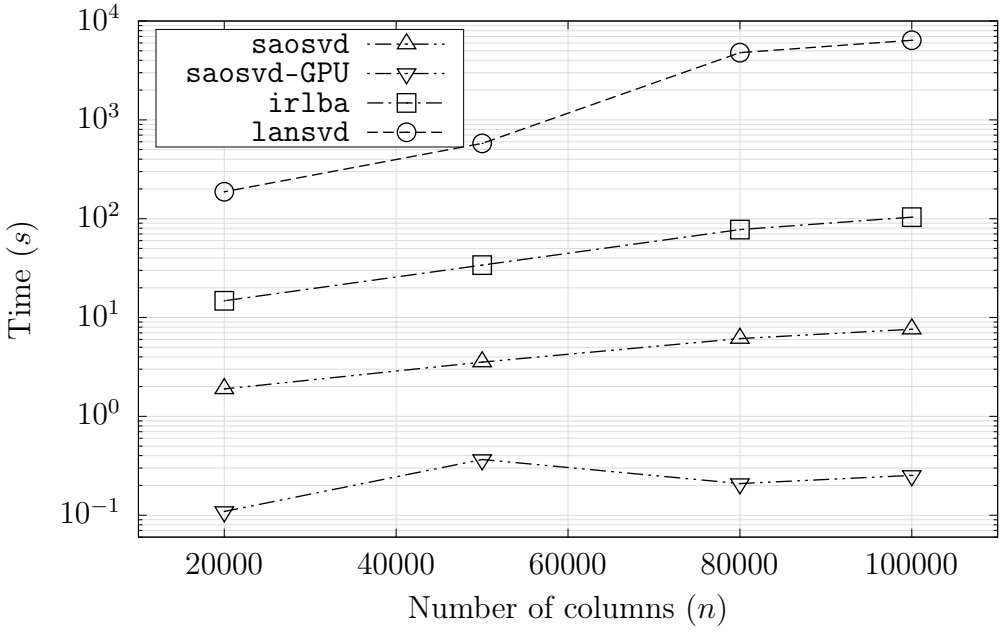


Figure 7.2: The decomposition times required for the multiplicate and slowly decaying singular values in Example 2, with $p = 1 \times 10^6$ and $\sigma = 12$. Both `svds` and `svds-GPU` failed to converge for all number of columns.

and $\sigma = 12$, while varying the number of columns-n of $\mathbf{A}$, to additionally test a solvers performance for dealing with rectangular matrices of differing size.

We present the decomposition time in Figure 7.2, with both the time and accuracy presented in Table 7.4. Both `svds` and `svds-GPU` failed to converge for all column sizes, clearly demonstrating a lack of robustness, which is again seen in examples to come. `saosvd` requires *significantly* lower time-complexity than the Lanczos solvers, by orders of magnitude, across all number of column sizes. This confirms that our method is far more resilient to multiplicity and slowly decaying singular values, with the Lanczos methods requiring substantially greater time-complexity. Of the solvers that did converge, all solvers correctly performed the respective decompositions.

7.8.3 Example 3: uniformly distributed sparse random matrices

Matrices with uniformly distributed random entries, $\mathbf{A} \in (0, 1)$, and singular values following a Marchenko-Pastur distribution [112], are constructed with $\sigma = 10$ and $p = n = 10^5$. The decomposition of random matrices is useful for many applications in random-matrix theory, of which the most important can be found in [32].

Random matrices are constructed for a chosen sparsity fraction defined in (7.39), using the MATLAB function `sprand`. An increase in the sparsity fraction γ results in singular values that decay slower, hence the difficulty of the decomposition increases with γ. The results for the CPU and GPGPU solvers for the sparse random matrix tests are presented in Table 7.5 and 7.6 respectively, with the decomposition times given in Figure 7.3.

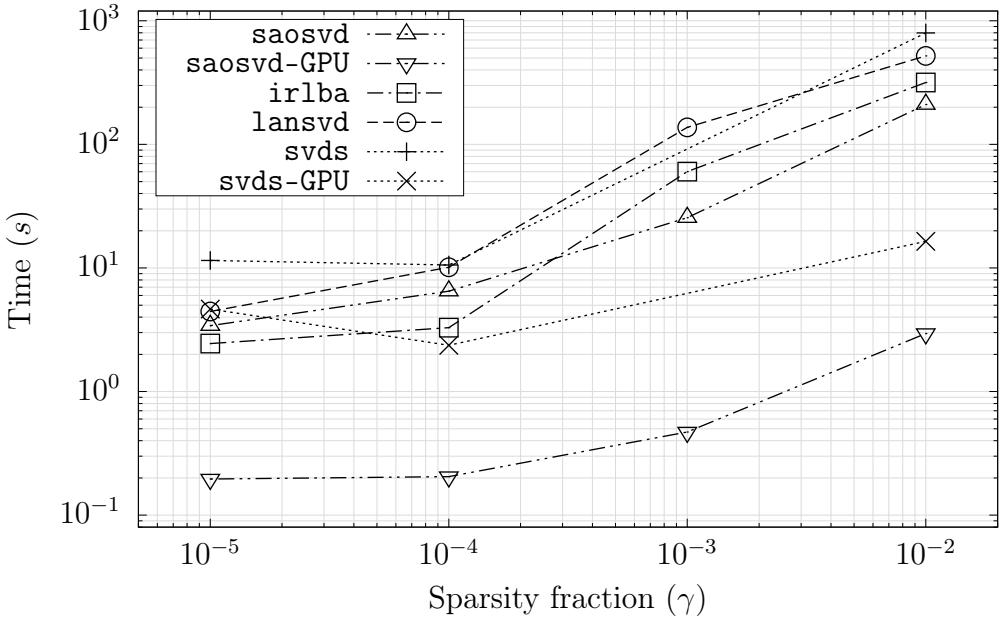


Figure 7.3: The decomposition times required for the random matrices in Example 3, with $p = n = 1 \times 10^6$ and $\sigma = 10$.

From Figure 7.3, it is clear the Lanczos methods perform better for lower values of γ, with `irlba` narrowly besting `saosvd`. For larger values of γ, `saosvd` outperforms all the Lanczos methods, again highlighting a resilience to slowly decaying singular values. For $\gamma = 10^{-3}$, both `svds` and `svds-GPU` failed the decomposition, again exposing the lack of robustness in MATLAB's `svds` solver.

Of the GPGPU bound solvers, `saosvd-GPU` requires significantly lower time complexity compared to `svds-GPU`, while still managing to correctly perform all decompositions. As seen in Example 1, `saosvd-GPU` again experiences a larger speedup over `saosvd` than `svds-GPU` manages over `svds`, demonstrating the effectiveness of `saosvd` on a parallel computational device.

7.8.4 Real-world datasets

To test the performance of all the solvers on real-world datasets, we select test matrices from the Harwell-Boeing and Florida sparse matrix collections, which contain varying dimensionality, sparsity and singular value distributions. The matrix properties and references, along with the relevant disciplines from which they arise, can be found in Table 7.1.

The results for the CPU bound solvers are presented in Table 7.7, while the results for the GPGPU accelerated solvers are presented in Table 7.8. Of the CPU bound methods, `saosvd` is both competitive and robust compared to the Lanczos methods, with the singular value distribution of a specific matrix leaning results in favor of different respective solvers. In terms of decomposition time, `saosvd` performs the highest number of fastest decompositions, with resilience to significantly high decomposition times for any given problem, which the

Lanczos methods are prone to.

Table 7.8 clearly shows that `saosvd-GPGPU` outperforms `svds-GPU`, both in terms of robustness as well as performing the decompositions with significantly lower time-complexity. Furthermore, it is evident that `saosvd-GPU` achieves a far greater speedup over `saosvd` than `svds-GPU` manages over `svds`, which is clearly shown in Figure 7.4. For perspective, the maximum speedup `svds-GPU` achieved over `svds` was only 7.15, while `saosvd-GPGPU` managed a maximum speedup of 88.36 over `saosvd`, confirming that `saosvd` is able to exploit GPGPU performance to a far greater degree than `svds`.

The NLPKKT160 test is worthy of special mention, since the large problem size highlights the prohibitively expensive memory requirements of Lanczos methods. The state-of-the-art PROPACK `lansvd` solver required more memory than our 128 GB machine could accommodate, despite the matrix storage requiring only a fraction of the system memory. Furthermore, `svds-GPU` required excessive memory on our 11 GB GPGPU, despite the matrix only requiring a fraction of the GPGPU memory for storage, highlighting the difficulty of implementing Lanczos methods on the limited memory devices available to us today. As a reference, we store both $\mathbf{A}$ as well as $\mathbf{A}^{\mathrm{T}}$ in CSR format in GPGPU memory; yet since we have *a priori known* memory requirements, we do not exceed the memory limit.

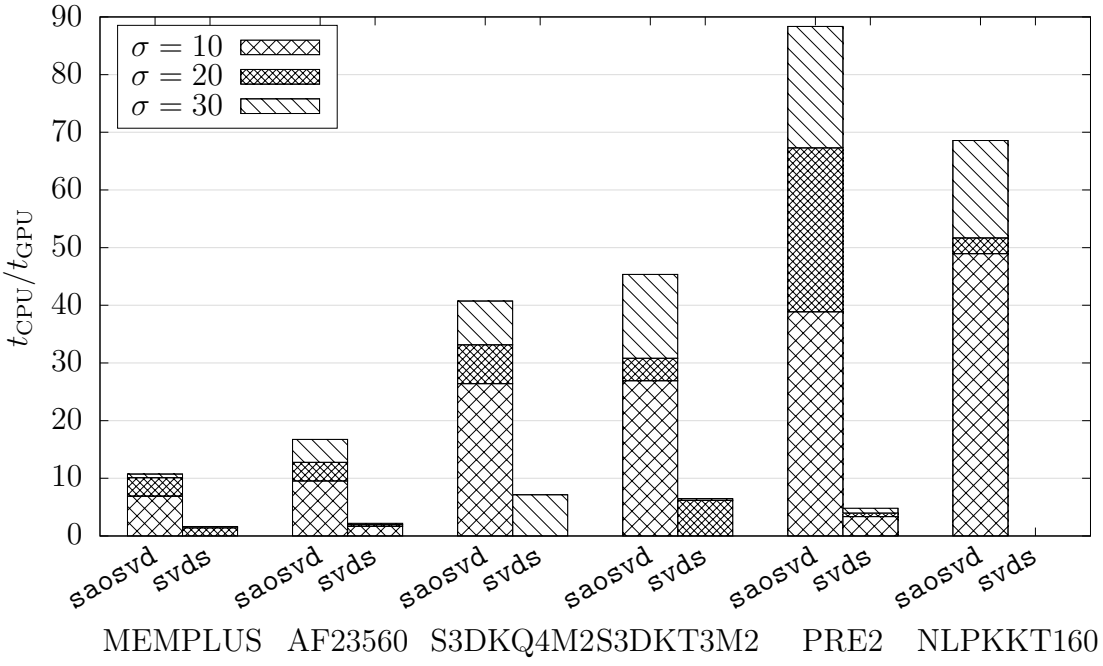


Figure 7.4: The speedup achieved for rank-σ decompositions, across the real-world datasets in Table 7.7, between CPU and GPGPU implementations of `saosvd` and `svds`.

7.9 Conclusion and recommendations

Our proposed method, `saosvd`, for incremental singular value decomposition, addresses the fundamental issues faced by Lanczos methods; namely `saosvd` is covariance-free, shows resilience to and converges under singular value multiplicity and requires constant memory.

`saosvd` solves a sequence of equality constrained quadratic-like approximations for each singular triplet, and is able to exploit cheap second-order information to significantly reduce the required time-complexity. Each quadratic-like approximation is solved for using a feasible descent method, with few iterations required for convergence. Furthermore, we introduce a novel scaling strategy for the objective function, which relies upon previously solved for singular value information, allowing `saosvd` to solve singular values of differing orders of magnitude.

Importantly, `saosvd` has the salient feature of closed-form primal and dual variable updates, both of which are embarrassingly parallel and suitable for implementation on massively parallel computational devices, such as GPGPUs. We demonstrate herein how effective `saosvd` is in a GPGPU context, compared to MATLAB's `svds` GPGPU implementation; where `saosvd` is able to achieve a far greater reduction in time-complexity, resulting in a solver with the lowest time-complexity for all problems tested in this study.

Lastly, we demonstrate the robustness and efficacy of `saosvd` compared to state-of-the-art Lanczos methods. `saosvd` does not require a parallel context to remain competitive with the Lanczos methods, which our numerical tests confirm across a wide range of large-scale test problems. `saosvd` is the only method to correctly perform all the decompositions in this study, demonstrating the robustness of our approach.

In future, we wish to develop a more efficient GPGPU implementation of `saosvd`, as well as investigate the effects of mixed precision computation, allowing for larger problem sizes to be tested in today's memory limited devices.

7.10 Tables for numerical results

Table 7.2: Numerical results for the variably decaying singular values in Example 1, with varying decomposition rank-σ and fixed matrix size $p = n = 1 \times 10^5$. († indicates failure of the decomposition step, despite the algorithm converging.)

	`saosvd`				`irlba`		`lansvd`		`svds`	
σ	N_f	N_l	κ (%)	t (s)	κ (%)	t (s)	κ (%)	t (s)	κ (%)	t (s)
12	47	1	4.36×10^{-6}	3.94×10^{0}	-3.56×10^{-7}	$\mathbf{7.24\times10^{-1}}$	-6.92×10^{-6}	1.09×10^{0}	-1.35×10^{-5}	1.62×10^{0}
16	70	3	-5.47×10^{-5}	5.60×10^{0}	-1.51×10^{-6}	$\mathbf{7.16\times10^{-1}}$	-5.88×10^{-5}	1.50×10^{0}	-7.95×10^{-6}	2.24×10^{0}
20	83	4	-6.54×10^{-1}	7.19×10^{0}	-5.73×10^{-1}	$\mathbf{1.07\times10^{0}}$	-7.26×10^{-1}	2.37×10^{0}	-5.64×10^{-1}	3.98×10^{0}
24	162	19	-4.60×10^{0}	1.20×10^{1}	$\dagger1.71\times10^{1}$	1.71×10^{0}	-3.40×10^{0}	5.30×10^{0}	-4.58×10^{0}	$\mathbf{3.51\times10^{0}}$

Table 7.3: Numerical results for the variably decaying singular values in Example 1 using GPGPU-accelerated solvers, with varying decomposition rank-σ and fixed matrix size $p = n = 1 \times 10^5$.

	`saosvd-GPU`					`svds-GPU`		
σ	N_f	N_l	κ (%)	t (s)	$t_{\text{CPU}}/t_{\text{GPU}}$	κ (%)	t (s)	$t_{\text{CPU}}/t_{\text{GPU}}$
12	42	3	-3.16×10^{-5}	$\mathbf{1.40\times10^{-1}}$	28.15	-1.26×10^{-6}	3.53×10^{-1}	4.58
16	58	4	-9.84×10^{-5}	$\mathbf{1.76\times10^{-1}}$	31.81	-6.79×10^{-6}	3.97×10^{-1}	5.63
20	71	6	-8.51×10^{-1}	$\mathbf{2.16\times10^{-1}}$	33.37	-9.39×10^{-1}	5.67×10^{-1}	7.02
24	85	9	-2.48×10^{0}	$\mathbf{2.52\times10^{-1}}$	47.69	-3.23×10^{0}	5.16×10^{-1}	6.81

Table 7.4: Numerical results for the multiplicate and slowly decaying singular values in Example 2, with varying column size-n, fixed row size $p = 1 \times 10^5$ and a fixed decomposition rank $\sigma = 12$. The speedup realized by the GPGPU implementation of `saosvd` is denoted by $S = t_{\text{CPU}}/t_{\text{GPU}}$, while both `svds` and `svds-GPU` failed to converge for all column sizes tested. (— indicates failure of an algorithm to converge.)

	`saosvd`				`irlba`		`lansvd`		`saosvd-GPU`				
n	N_f	N_l	κ (%)	t (s)	κ (%)	t (s)	κ (%)	t (s)	N_f	N_l	κ (%)	t (s)	S
1×10^5	132	25	-4.19×10^0	$\mathbf{7.60\times10^0}$	-4.16×10^0	1.04×10^2	-4.18×10^0	6.38×10^3	142	54	-4.22×10^0	2.53×10^{-1}	30.10
8×10^4	126	35	-4.19×10^0	$\mathbf{6.12\times10^0}$	-4.15×10^0	7.75×10^1	-4.19×10^0	4.78×10^3	128	42	-4.22×10^0	2.09×10^{-1}	29.23
5×10^4	121	24	-4.19×10^0	$\mathbf{3.54\times10^0}$	-4.14×10^0	3.39×10^1	-4.21×10^0	5.77×10^2	395	156	-4.66×10^0	3.66×10^{-1}	9.68
2×10^4	151	43	-4.07×10^0	$\mathbf{1.89\times10^0}$	-4.06×10^0	1.48×10^1	-4.15×10^0	1.87×10^2	138	54	-4.11×10^0	1.09×10^{-1}	17.35

Table 7.5: Numerical results for the random sparse matrices in Example 3, with a fixed matrix size $p = n = 1 \times 10^5$, $\sigma = 10$ and varying sparsity fraction γ.

	`saosvd`				`irlba`		`lansvd`		`svds`	
γ	N_f	N_l	κ (%)	t (s)	κ (%)	t (s)	κ (%)	t (s)	κ (%)	t (s)
1×10^{-5}	324	127	-7.67×10^0	3.41×10^0	-7.37×10^0	$\mathbf{2.45\times10^0}$	-7.19×10^0	4.45×10^0	-7.43×10^0	1.15×10^1
1×10^{-4}	332	146	-7.67×10^0	6.49×10^0	-7.63×10^0	$\mathbf{3.29\times10^0}$	-7.74×10^0	1.01×10^1	-7.67×10^0	1.06×10^1
1×10^{-3}	325	83	-5.93×10^0	$\mathbf{2.55\times10^1}$	-5.95×10^0	6.02×10^1	-6.02×10^0	1.37×10^2	—	—
1×10^{-2}	355	105	-5.89×10^0	$\mathbf{2.10\times10^2}$	-5.98×10^0	3.17×10^2	-5.94×10^0	5.20×10^2	-5.93×10^0	7.98×10^2

Table 7.6: GPGPU numerical results for the random sparse matrices in Example 3, with a fixed matrix size $p = n = 1 \times 10^5$, $\sigma = 10$ and varying sparsity fraction γ.

	`saosvd-GPU`					`svds-GPU`		
γ	N_f	N_l	κ (%)	t (s)	$t_{\text{CPU}}/t_{\text{GPU}}$	κ (%)	t (s)	$t_{\text{CPU}}/t_{\text{GPU}}$
1×10^{-5}	371	190	-7.62×10^0	1.96×10^{-1}	17.41	-7.62×10^0	4.65×10^0	2.48
1×10^{-4}	341	129	-7.70×10^0	2.05×10^{-1}	31.67	-7.54×10^0	2.37×10^0	4.47
1×10^{-3}	303	145	-5.89×10^0	4.70×10^{-1}	54.20	—	—	—
1×10^{-2}	347	112	-5.97×10^0	2.95×10^0	71.24	-5.93×10^0	1.64×10^1	48.58

Table 7.7: Numerical results for the large-scale real-world datasets in Example 4, using CPU based solvers. (— indicates that the algorithm failed to converge.)

		saosvd				irlba		lansvd		svds	
Matrix	σ	N_f	N_l	κ (%)	t (s)	κ (%)	t (s)	κ (%)	t (s)	κ (%)	t (s)
MEMPLUS	10	154	43	-1.09×10^{0}	4.39×10^{-1}	-1.03×10^{0}	$\mathbf{2.03\times10^{-1}}$	-1.17×10^{0}	5.71×10^{0}	—	—
	20	386	112	-1.36×10^{0}	$\mathbf{1.67\times10^{0}}$	-1.35×10^{0}	2.02×10^{0}	-1.32×10^{0}	7.95×10^{0}	-1.33×10^{0}	6.39×10^{0}
	30	637	205	-1.85×10^{0}	3.37×10^{0}	-1.54×10^{0}	2.21×10^{0}	-1.73×10^{0}	5.86×10^{0}	-1.89×10^{0}	$\mathbf{2.11\times10^{0}}$
AF23560	10	170	40	-1.54×10^{0}	6.60×10^{-1}	-1.48×10^{0}	$\mathbf{1.73\times10^{-1}}$	-1.33×10^{0}	2.66×10^{-1}	-1.54×10^{0}	4.27×10^{-1}
	20	375	98	-2.56×10^{0}	2.15×10^{0}	-2.23×10^{0}	4.63×10^{-1}	-2.60×10^{0}	$\mathbf{4.26\times10^{-1}}$	-2.17×10^{0}	5.04×10^{-1}
	30	714	191	-2.32×10^{0}	4.66×10^{0}	-2.23×10^{0}	6.95×10^{-1}	-2.49×10^{0}	$\mathbf{5.03\times10^{-1}}$	-2.84×10^{0}	8.33×10^{-1}
S3DKQ4M2	10	367	91	-5.55×10^{0}	$\mathbf{8.19\times10^{0}}$	-5.37×10^{0}	1.35×10^{1}	-5.38×10^{0}	1.55×10^{1}	—	—
	20	775	226	-5.30×10^{0}	$\mathbf{2.06\times10^{1}}$	-5.45×10^{0}	7.75×10^{1}	-5.45×10^{0}	2.38×10^{1}	—	—
	30	1120	360	-5.19×10^{0}	3.70×10^{1}	-5.35×10^{0}	7.74×10^{1}	-5.32×10^{0}	$\mathbf{2.90\times10^{1}}$	-5.41×10^{0}	3.89×10^{1}
S3DKT3M2	10	372	118	-4.38×10^{0}	$\mathbf{7.30\times10^{0}}$	-4.22×10^{0}	1.25×10^{1}	-4.25×10^{0}	1.47×10^{1}	—	—
	20	707	242	-4.26×10^{0}	$\mathbf{1.80\times10^{1}}$	-4.30×10^{0}	7.99×10^{1}	-4.35×10^{0}	2.08×10^{1}	-4.18×10^{0}	5.70×10^{1}
	30	1165	446	-4.15×10^{0}	3.98×10^{1}	-4.20×10^{0}	8.85×10^{1}	-4.09×10^{0}	3.21×10^{1}	-4.11×10^{0}	$\mathbf{2.72\times10^{1}}$
PRE2	10	176	58	-2.65×10^{0}	1.60×10^{1}	-2.69×10^{0}	$\mathbf{1.48\times10^{1}}$	-2.59×10^{0}	4.52×10^{1}	-3.30×10^{0}	5.44×10^{1}
	20	357	107	-3.19×10^{0}	6.81×10^{1}	-3.33×10^{0}	5.66×10^{2}	-4.26×10^{0}	4.77×10^{1}	-3.93×10^{0}	$\mathbf{2.27\times10^{1}}$
	30	533	190	-2.09×10^{0}	1.47×10^{2}	-2.24×10^{0}	7.98×10^{1}	-1.63×10^{0}	7.14×10^{1}	-1.83×10^{0}	$\mathbf{4.59\times10^{1}}$
NLPKKT160	10	475	144	-7.92×10^{0}	$\mathbf{7.85\times10^{2}}$	-7.92×10^{0}	2.72×10^{3}	—	—	—	—
	20	876	236	-7.87×10^{0}	$\mathbf{2.23\times10^{3}}$	-7.87×10^{0}	8.31×10^{3}	—	—	—	—
	30	1241	463	-7.79×10^{0}	$\mathbf{4.61\times10^{3}}$	-7.79×10^{0}	1.10×10^{4}	—	—	-7.79×10^{0}	6.57×10^{3}

Table 7.8: Numerical results for the large-scale real-world datasets in Example 4, using GPGPU-accelerated solvers, in Example 4. $t_{\text{CPU}}/t_{\text{GPU}}$ denotes the factor reduction in time-complexity compared to the CPU counterpart of the solver. (— indicates that the algorithm failed to converge.)

		saosvd-GPU					svds-GPU		
Matrix	σ	N_f	N_l	κ (%)	t (s)	$t_{\text{CPU}}/t_{\text{GPU}}$	κ (%)	t (s)	$t_{\text{CPU}}/t_{\text{GPU}}$
MEMPLUS	10	142	25	-1.25×10^{0}	$\mathbf{6.34\times10^{-2}}$	6.92	—	—	—
	20	383	160	-1.38×10^{0}	$\mathbf{1.66\times10^{-1}}$	10.11	-1.24×10^{0}	4.54×10^{0}	1.41
	30	624	263	-2.08×10^{0}	$\mathbf{3.14\times10^{-1}}$	10.73	-1.62×10^{0}	1.27×10^{0}	1.65
AF23560	10	161	33	-2.20×10^{0}	$\mathbf{6.91\times10^{-2}}$	9.55	-2.68×10^{0}	2.51×10^{-1}	1.70
	20	394	106	-2.05×10^{0}	$\mathbf{1.69\times10^{-1}}$	12.75	-3.63×10^{0}	2.56×10^{-1}	1.97
	30	654	216	-1.86×10^{0}	$\mathbf{2.79\times10^{-1}}$	16.74	-2.43×10^{0}	3.80×10^{-1}	2.19
S3DKQ4M2	10	387	127	-5.45×10^{0}	$\mathbf{3.10\times10^{-1}}$	26.41	—	—	—
	20	798	286	-5.35×10^{0}	$\mathbf{6.22\times10^{-1}}$	33.16	—	—	—
	30	1092	391	-5.34×10^{0}	$\mathbf{9.09\times10^{-1}}$	40.73	-5.46×10^{0}	5.44×10^{0}	7.15
S3DKT3M2	10	366	120	-4.41×10^{0}	$\mathbf{2.71\times10^{-1}}$	26.91	—	—	—
	20	729	343	-4.23×10^{0}	$\mathbf{5.83\times10^{-1}}$	30.80	-4.17×10^{0}	9.20×10^{0}	6.20
	30	1098	452	-4.22×10^{0}	$\mathbf{8.78\times10^{-1}}$	45.35	-4.17×10^{0}	4.20×10^{0}	6.46
PRE2	10	161	37	-2.36×10^{0}	$\mathbf{4.13\times10^{-1}}$	38.88	-2.89×10^{0}	1.62×10^{1}	3.35
	20	339	154	-4.13×10^{0}	$\mathbf{1.01\times10^{0}}$	67.30	-3.81×10^{0}	5.73×10^{0}	3.97
	30	533	226	-2.09×10^{0}	$\mathbf{1.67\times10^{0}}$	88.36	-2.02×10^{0}	9.51×10^{0}	4.82
NLPKKT160	10	405	262	-7.92×10^{0}	$\mathbf{1.60\times10^{1}}$	48.95	—	—	—
	20	913	843	-7.87×10^{0}	$\mathbf{4.32\times10^{1}}$	51.69	—	—	—
	30	1184	1261	-7.79×10^{0}	$\mathbf{6.72\times10^{1}}$	68.57	—	—	—

Chapter 8

Softening the no-free-lunch theorems for structural optimization

The work presented here originates from a paper titled "Softening the no-free-lunch theorems for structural optimization" [113], which has been submitted at the time of writing. The paper is co-authored by Prof. Albert A. Groenwold.

8.1 Abstract

We demonstrate a softening of the no-free-lunch (NFL) theorems for optimization, by simultaneously executing multiple solvers on a multi-core machine, for a range of large-scale structural problems; as opposed to using a single solver on a multi-core machine. Since each solver exploits a different solution strategy, the solver performance is highly dependent on problem type, all in the spirit of NFL. The solution strategies exploited include pure dual, interior-point, active-set and augmented Lagrangian methods. All algorithms solve a sequence of convex and separable quadratic-like problems, able to capture both reciprocal and exponential-like behavior, which is desirable in structural optimization.

Numerical results are demonstrated for challenging structural problems, containing up to tens of millions primal variables and constraints, run on a single quad-core CPU with 128 GB of RAM. Although the solution time for a given problem, using a single solver, may be compromised in this approach, the overall solution time for the entire test set is significantly lower, hence we soften NFL. The state-of-the-art solvers tested herein include Galahad `LSQP` and IBM ILOG `CPLEX`, together with `ALGENCAN` and the popular Falk dual.

8.2 Introduction

The no-free-lunch (NFL) theorems for optimization [65] unequivocally dictate that all algorithms, on average, perform identically across all possible problem variations. Although possibly surprising, the implications of NFL are such that no state-of-the-art algorithm will outperform a *random* search across all possible problem types [65]. Furthermore, improving

the efficacy of an algorithm for a given problem type results in the algorithm performing worse, on average, over the remainder of possible problem types. Of course, *a priori* knowledge of a problem may be exploited by a suitable algorithm that takes advantage of said knowledge; however, the nature of 'black-box' sensitivity analysis required by many structural problems often gives no indication of such knowledge. 'Black-box' problems are common in computational fluid dynamics (CFD), finite element methods (FEM), partial differential equations (PDEs), etc. Even for problems not requiring some 'black-box', obtaining optimization specific knowledge may still be difficult, since the solution point is often unknown. Continuously selecting the best-suited algorithm for differing problem types is thus, in the spirit of NFL, impossible. Indeed, it is this challenge which we wish to address herein, accepting that no single algorithm will be suitable for problems where sufficient *a priori* knowledge is unavailable.

With the advent of multi-core computer architectures, the parallelism prevalent in many basic linear algebra operations is readily exploited by suitable algorithms. Although the total computational complexity is unchanged in said algorithms, the time-complexity required may be significantly lower on multi-core machines, when compared to single-core variations. In our proposed approach, we sacrifice the computational resources afforded to such algorithms[1], instead allocating resources to separate solver algorithms. We do this to soften the effects of NFL, since the failure of a single algorithm on a given problem may be prohibitively expensive, as opposed to sharing resources amongst various solvers on a multi-core system. As we will show in what is to come, different solvers for a given problem often require orders of magnitude difference in solution time. Hence, using a single solver for a diverse range of problems can be prohibitively expensive, since it is likely that a different solver will fare far better for at least one problem in the given range. Furthermore, we demonstrate that sharing resources between solvers, on our modest system, results in a maximum twofold increase in the standalone solver solution time. Since a twofold increase is far less expensive than the orders of magnitude observed between different solvers, the NFL effects are softened over our selected test problems, where no single solver is fastest in solving all the problems. Naturally, more resources will allow a given solver to exploit parallelism to a greater degree, but we purposefully demonstrate a suboptimal scenario, by means of a single quad-core CPU executing five different solvers simultaneously. Many structural optimization applications require expensive sensitivity analyses, hence solvers requiring fewer sensitivity analyses often converge first. Indeed, our proposed approach can be applied to said applications, provided sufficient resources are available to perform simultaneous sensitivity analyses.

To the authors knowledge, no studies of softening NFL in structural optimization exist, especially for challenging high-dimensional convex problems. A benefit of our proposed approach is the relaxation of a global convergence mechanism, since it is likely that a few solvers will achieve local convergence to the optimum of the convex subproblem , under the assumption of a reasonable starting point and sufficient solver diversity. The aforementioned is important, since although the subproblems are convex, (local) convergence is not guaranteed in the absence of a global convergence mechanism. In the spirit of NFL, introducing such global mechanisms may be computationally expensive, hence we wish to rely on local subproblem convergence. Of course, our solver execution strategy could be extended to include solvers

[1] All the algorithms tested herein can exploit multi-core architectures.

with and without global convergence mechanisms. To assess the effects of enforcing global convergence, we provide an example problem with and without global convergence enforced, demonstrated in sections to come.

We depart with a constrained nonlinear problem $\mathcal{P}_{\text{NLP}}$, given by

$$\begin{aligned} \min_{\mathbf{x}} \ & f_0(\mathbf{x}) \\ \text{subject to} \ & f_j(\mathbf{x}) = 0, && j = 1, \ldots, m', \\ & f_j(\mathbf{x}) \leq 0, && j = m'+1, \ldots, m, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, && i = 1, \ldots, n, \end{aligned} \tag{8.1}$$

where $\mathbf{x} \in \mathcal{R}^n$ represents the n primal (design) variables. The mappings $f_0(\mathbf{x}) : \mathcal{R}^n \to \mathcal{R}$ and $f_j(\mathbf{x}) : \mathcal{R}^n \to \mathcal{R}$, $j = 1, \ldots, m$ denote the first-order continuous objective and constraint functions respectively, where m' and $m - m'$ denote the number of equality and inequality constraints. The upper and lower primal bounds, $\hat{x}_i$ and $\check{x}_i$, ensures that the primal variables reside in a closed and bound set $\mathbf{x} \in \mathcal{X} \subset \mathcal{R}^n$, which is a requirement for many of the algorithms tested herein.

In structural optimization, sequential approximate optimization (SAO) is arguably state-of-the-art in solving $\mathcal{P}_{\text{NLP}}$. SAO solves an iterative sequence of *separable* and *convex* approximate quadratic-like problems $\mathcal{P}_{\text{P}}[k]$, where each $k = 0, 1, 2, \ldots$, requires a sensitivity analysis to construct $\mathcal{P}_{\text{P}}[k]$. Consider the approximate problem $\mathcal{P}_{\text{P}}[k]$ at an iteration k, where

$$\begin{aligned} \min_{\mathbf{x}} \ & \tilde{f}_0^k(\mathbf{x}) \\ \text{subject to} \ & \tilde{f}_j^k(\mathbf{x}) = 0, && j = 1, \ldots, m', \\ & \tilde{f}_j^k(\mathbf{x}) \leq 0, && j = m'+1, \ldots, m, \\ & \check{x}_i^k \leq x_i \leq \hat{x}_i^k, && i = 1, \ldots, n. \end{aligned} \tag{8.2}$$

As previously mentioned, the approximate primal problem in (8.2) is constructed from separable functions, such that

$$\tilde{f}_j(\mathbf{x}) = \sum_{i=1}^{n} \tilde{f}_j(x_i), \qquad j = 0, \ldots, m, \tag{8.3}$$

with many algorithms able to efficiently exploit separable problems. The separability of $\mathcal{P}_{\text{P}}[k]$ is a result of approximate *diagonal* Hessian information, requiring only $\mathcal{O}(n)$ Hessian elements to be stored in memory and computed. The computation of the diagonal Hessian elements are cheap, since the Hessian elements are simple first-order Taylor series constructed around the approximate functions, hence they have analytical solutions. In structural optimization, the diagonal Hessian elements are so-called direct or intervening variables; the latter able to exploit reciprocal and exponential-like behavior, such as the inverse relationship between stress and area, i.e. $z_i = x_i^{-1}$ [114]. Furthermore, intervening variables are always uncoupled, ensuring the formation of a diagonal Hessian. Direct variables are often suited to more general problems, where intervening variables cannot be exploited.

Some popular algorithms which exploit the structure in (8.3) include Fleury and Braibant's convex linearization algorithm (CONLIN) [10], as well as Svanberg's method of moving

asymptotes (MMA) [8, 9]. However, throughout this study, we strictly make use of Groenwold and Etman's SAOi framework [11] for large-scale optimization. SAOi relies upon so-called 'approximated-approximations' [38], wherein quadratic approximations are constructed to any number of popular Hessian approximations, including the reciprocal and exponential-like approximations. Hence, the convex SAOi subproblems in (8.2) may be solved with any suitable optimization algorithm, which in turn forms the basis for this study.

The solution of the SAOi subproblems are subject to NFL, despite being comprised of a simple, separable and convex structure. Popular approaches used to solve for the subproblems in structural optimization include pure dual methods [8, 9, 10, 74], Lagrangian methods [71] and augmented Lagrangian methods [35]. Lagrangian methods, when cast into a sequential quadratic programming (SQP) framework, are further differentiated into two main classes, namely (barrier) interior-point and active-set methods.

To *a priori* select the best of the above mentioned strategies for a given problem is, due to NFL, impossible. Traditional selection methods mostly rely on some heuristic, one example being the use of pure dual methods when $m \ll n$. Notwithstanding that pure dual methods pose no theoretical difficulties for convex and separable subproblem solutions[2], the solution space is only in $\mathcal{R}^m$, since the primal variables are updated using closed-form expressions. Hence, pure dual methods can be extremely efficient when $m \ll n$, with one such popular example being the Falk dual [70]. However, even for problems where $m \ll n$, pure dual methods may still be outperformed by SQP methods; a somewhat surprising result we demonstrate in sections to come.

Our paper makes the following contributions: We propose a multi-solver, multi-core execution strategy to soften the NFL effects observed in structural optimization. Our proposed strategy significantly reduces the *mean* solution time required for a wide-range of structural test problems, where no single solver was able to successfully solve all problems. Despite all the solvers sharing computational resources on a single quad-core CPU, we further demonstrate that the performance impact compared a single solver implementation is not severe, notwithstanding that many of the problems contain tens of millions of primal variables and/or constraints.

Our paper is arranged as follows: In Section 8.3, we introduce the separable quadratic-like approximations exploited by all solvers, followed by sequential quadratic programming (SQP) and pure dual problem statements. Numerical results for a wide range of challenging, large-scale structural problems are demonstrated in Section 8.5, before finally presenting conclusions and recommendations for future work in Section 8.6.

8.3 Sequential approximate optimization

In SAO, the iterative solution for subproblems $\mathcal{P}_{\mathrm{P}}[k]$, $k = 1, 2, \ldots$, is required, yielding a solution point $\mathbf{x}^{k+1}$ at each iteration k, around which a new problem $\mathcal{P}_{\mathrm{P}}[k+1]$ is constructed.

[2]Provided a suitable solver is used that can handle bound constraints and second-order discontinuities, such as `l-BFGS-b` [4].

The convergence of the SAO sequence is well-established, provided a suitable mechanism is used to enforce global convergence. Notwithstanding that none of our test problems required a global convergence mechanism, since at least one solver was able to achieve local convergence for any given problem, we elaborate on suitable SAO global enforcement mechanisms in Section 8.4, followed by examples in Section 8.5.

8.3.1 Separable quadratic-like approximations

For notational simplicity and brevity in what is to follow, consider the following notation

$$\begin{aligned} f_j^k &= f_j(\mathbf{x}^k), \\ \left(\frac{\partial f_j}{\partial x_i}\right)^k &= \frac{\partial f_j(\mathbf{x}^k)}{\partial x_i}, \\ \boldsymbol{\nabla} f_j^k &= \left[\left(\frac{\partial f_j}{\partial x_1}\right)^k, \ldots, \left(\frac{\partial f_j}{\partial x_n}\right)^k\right]^{\mathrm{T}}, \end{aligned}$$

where $\boldsymbol{\nabla}$ is the primal differential operator. Using the aforementioned notation, the separable and convex approximate functions are constructed using an incomplete series expansion (ISE) [2], such that

$$\tilde{f}_j^k(\mathbf{s}) = f_j^k + \boldsymbol{\nabla} f_j^{k\mathrm{T}} \mathbf{s} + \mathbf{s}^{\mathrm{T}} \mathbf{C}_j^k \mathbf{s}, \tag{8.4}$$

where $\mathbf{s} = (\mathbf{x} - \mathbf{x}^k)$ and $\mathbf{C}_j^k \in \mathcal{R}^{n \times n}$ a diagonal matrix containing approximate second-order information of $\tilde{f}_j^k(\mathbf{x})$. The diagonal elements used to form $\mathbf{C}_j^k$, denoted by $c_{2i_j}^k \ \forall \ i = 1, \ldots, n$, are the direct or intervening variables, with the latter able to capture reciprocal and exponential-like behavior. Although many approximations for $c_{2i_j}^k$ exist [38], we shall herein consider but three instances, namely:

1. Snyman and Hay's spherical quadratic approximation (SPH-QDR) [54], wherein the curvature $c_{2i_j}^k = c_{2_j}^k \ \forall \ i$ is chosen such that $f_j^{k-1} = \tilde{f}_j^{k-1}$.
2. A non-spherical approximation, based on components of the first-order objective and constraint function information (GRAD), presented in [115].
3. The quadratic approximation to the reciprocal approximation [38] (T2:R), which is akin to the popular MMA [8] approximations.

Since the approximate functions are separable, each function in (8.4) can be expressed as the sum of n separable primal terms, given by

$$\tilde{f}_j^k(\mathbf{x}) = f_j^k + \sum_{i=1}^{n} \left(\frac{\partial f_j}{\partial x_i}\right)^k (x_i - x_i^k) + \sum_{i=1}^{n} c_{2i_j}^k (x_i - x_i^k)^2, \tag{8.5}$$

which forms the basis for the approximate problem statements, discussed in what is to follow.

8.3.2 Diagonal QP subproblems

Following the construction of the approximate quadratic-like functions in (8.4), we trivially form a sequential quadratic program (SQP) $\mathcal{P}_{\mathrm{PQ}}[k]$ from $\mathcal{P}_{\mathrm{P}}[k]$, as

□ *Quadratic approximate program* $\mathcal{P}_{\mathrm{PQ}}[k]$

$$\begin{aligned} \min_{\mathbf{s}} \ & \tilde{f}_0^k(\mathbf{s}) = f_0^k + \boldsymbol{\nabla} f_0^{k\mathrm{T}}\mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}\mathbf{Q}^k\mathbf{s} \\ \text{subject to} \ & \tilde{f}_j^k(\mathbf{s}) = f_j^k + \boldsymbol{\nabla} f_j^{k\mathrm{T}}\mathbf{s} = 0, && j = 1, \ldots, m', \\ & \tilde{f}_j^k(\mathbf{s}) = f_j^k + \boldsymbol{\nabla} f_j^{k\mathrm{T}}\mathbf{s} \leq 0, && j = m'+1, \ldots, m, \\ & \check{x}_i^k \leq x_i \leq \hat{x}_i^k, && i = 1, \ldots, n. \end{aligned} \tag{8.6}$$

In $\mathcal{P}_{\mathrm{PQ}}[k]$, the diagonal matrix $\mathbf{Q}^k$ is the Hessian of the approximate Lagrangian $\tilde{\mathcal{L}}^k$. Since the dual variables at the solution point, $\boldsymbol{\mu}^{k*}$, are unknown, the dual variables at the previous iteration point, $\boldsymbol{\mu}^k$, are used to construct a *constant* Hessian, given by

$$Q_{ii}^k = \max\left(\epsilon,\ c_{2i_0}^k + \sum_{j=1}^{m} \mu_j^k c_{2i_j}^k\right), \quad i = 1, \ldots, n. \tag{8.7}$$

To ensure that the approximate Hessian for $\mathcal{P}_{\mathrm{PQ}}[k]$ is positive-definite, we simply require that each individual element of $\mathbf{Q}^k$ is positive, hence $\epsilon > 0$ with some small prescribed magnitude, say 10^{-6}. For more details on the construction of approximate SQP problems, the interested reader is referred to the work conducted by Etman *et al.* [52, 53].

Herein, we shall not prescribe on how $\mathcal{P}_{\mathrm{PQ}}[k]$ is solved. Instead, our chosen QP solvers, namely Galahad `LSQP` [51], IBM ILOG `CPLEX` [116] and `SLA` [69], are simply given $\mathcal{P}_{\mathrm{PQ}}[k]$ at each iteration. However, the solution strategies differ between the solvers, since `LSQP` and `CPLEX` favor an interior-point method, whereas `SLA` exploits a highly efficient active-set strategy. Importantly, all the QP solvers are *tailor-made* for separable quadratic problems in the form of $\mathcal{P}_{\mathrm{PQ}}[k]$, resulting in efficient subproblem solutions for large-scale problems.

8.3.3 Pure dual subproblems

The pure dual subproblems result from a second-order cone problem (SOCP) statement, where the approximate dual problem $\mathcal{P}_{\mathrm{D}}[k]$ is constructed, such that

□ *Dual approximate problem* $\mathcal{P}_{\mathrm{D}}[k]$

$$\begin{aligned} \min_{\mathbf{s}} \ & \tilde{f}_0^k(\mathbf{s}) = f_0^k + \boldsymbol{\nabla} f_0^{k\mathrm{T}}\mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}\mathbf{C}_0^k\mathbf{s} \\ \text{subject to} \ & \tilde{f}_j^k(\mathbf{s}) = f_j^k + \boldsymbol{\nabla} f_j^{k\mathrm{T}}\mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}\mathbf{C}_j^k\mathbf{s} = 0, && j = 1, \ldots, m', \\ & \tilde{f}_j^k(\mathbf{s}) = f_j^k + \boldsymbol{\nabla} f_j^{k\mathrm{T}}\mathbf{s} + \frac{1}{2}\mathbf{s}^{\mathrm{T}}\mathbf{C}_j^k\mathbf{s} \leq 0, && j = m'+1, \ldots, m, \\ & \check{x}_i^k \leq x_i \leq \hat{x}_i^k, && i = 1, \ldots, n. \end{aligned} \tag{8.8}$$

which allows for the construction of Falk's dual problem [70], given by

$$\begin{aligned} \max_{\boldsymbol{\mu}} \ & \tilde{\gamma}(\boldsymbol{\mu}) = \tilde{f}_0^k(\mathbf{x}(\boldsymbol{\mu})) + \sum_{j=1}^{m} \tilde{f}_j^k(\mathbf{x}(\boldsymbol{\mu}))\mu_j, \\ \text{subject to} \quad & \mu_j \geq 0, \qquad j = m' + 1, \ldots, m, \\ & \mathbf{x}(\boldsymbol{\mu}) \in \mathcal{X}. \end{aligned} \tag{8.9}$$

A salient feature of $\mathcal{P}_{\mathrm{D}}[k]$ is that the primal minimizers are available as closed-form expressions, as noted by Falk [70], since $\mathcal{P}_{\mathrm{D}}[k]$ is comprised of separable and convex functions. The closed-form primal minimizer is a function of the dual variables, where

$$x_i(\boldsymbol{\mu}) = \Pi \left(x_i^k - \left(\left(\frac{\partial f_0}{\partial x_i} \right)^k + \sum_{j=1}^{m} \mu_j \left(\frac{\partial f_j}{\partial x_i} \right)^k \right) \left(c_{2i_0}^k + \sum_{j=1}^{m} \mu_j c_{2i_j}^k \right)^{-1} \right), \tag{8.10}$$

and $\Pi(\cdot)$ is the projection onto $[\check{x}_i^k, \hat{x}_i^k]$, ensuring the primal bounds are respected. For the exact details on how we solve the Falk dual formulation given by $\mathcal{P}_{\mathrm{D}}[k]$, the interested reader is referred to Groenwold's work in [71]. Furthermore, we trivially bound the dual as a form of relaxation for infeasible subproblems, described in References [6, 83].

Although the Falk dual is attractive when the number of constraints is low, Fleury notes that the topology of the dual space formed by $\mathcal{P}_{\mathrm{D}}[k]$ is 'piecewise quadratic'; an implication of incorporating the primal bounds inside the dual problem [76]. Indeed, this can result in some problems requiring prohibitively expensive subproblem solution effort, especially in a high dimensional dual space.

8.4 Global convergence

As previously mentioned, all the algorithms tested herein are implemented within an existing SAO framework, namely SAOi [11]. SAOi enforces global convergence through one of three mechanisms, which are independent of solver choice. Hence, each solver can exploit any of three mechanisms, given by:

1. A trust-region method together with a non-linear acceptance filter, detailed in [79, 80, 81, 82]. Arguably, trust-region acceptance filters are today considered state-of-the-art in SQP-like methods, since the troublesome penalty parameters found in merit filters can be avoided.

2. Conservative convex separable Hessian approximations [78], based on the conservatism first proposed by Svanberg [62]. In [78], it was proven that only the active constraints required conservative approximations. For problems with many inactive constraints, using [78] can significantly reduce the required computational effort.

3. Filtered conservatism, which incorporates the salient features of both conservatism and the above mentioned trust-region method [1].

The interested reader is referred to [1] for an in-depth comparison between all three global convergence mechanisms, within the same SAO framework outlined in Section 8.3. To demonstrate the effect of a global convergence mechanism, we present two examples of a single problem in Section 8.5, relying upon local convergence and filtered-conservatism respectively.

8.5 Numerical experiments

For a given problem, an executable was compiled for each of the Galahad `LSQP` [51], IBM `CPLEX` [116], `SLA` [69], Falk dual[3] and `ALGENCAN` solvers [117]. `ALGENCAN` solves the $\mathcal{P}_{\text{PQ}}[k]$ problem in (8.6) using an augmented Lagrangian formulation, which proved to be useful in previous work on large-scale non-convex SAO subproblems [118]. All five executables were then simultaneously launched using GNU parallel [119], on a single quad-core (hyper-threaded) Intel Xeon-4112 CPU, paired with 128 GB[4] of 2666 MHz DDR4 system memory. FORTRAN f90, compiled with gfortran-9 and the -O3 optimization flag, was used to create the executables, all running under Ubuntu 18.04. All solvers made use of default settings, unless otherwise stated.

For the problem convergence criteria, let $\|\mathbf{x}-\mathbf{x}^k\|_2$ denote the primal norm, $\mathcal{K}$ the Euclidean norm of the KKT conditions, h the maximum constraint violation and N_f as the number of iterations, k, required for convergence. Problem convergence was satisfied when $h \leq 10^{-4}$ and either $\mathcal{K} \leq 10^{-4}$, or $\|\mathbf{x}-\mathbf{x}^k\|_2 \leq 10^{-5}$, was met. As is common in structural optimization, a move limit of 2×10^{-1} was used across all problems. The dual variables were bounded by a maximum value of 10^8 to ensure numerical stability, as well as to provide relaxation for the Falk dual. In the first iteration when no historical information was available, we initialized the dual variables such that $\boldsymbol{\mu}^0 = 0$. A maximum iteration count of $k = 10^4$, as well as a time limit of 4 hours, was placed on all problems, with '—' and † used in the tables throughout to indicate violations of the respective limits. Lastly, * denotes the respective values after problem termination.

Despite Wolpert and Macready presenting measurements techniques for NFL [65], we chose to measure subproblem solver solution time. A problem was terminated when any of the solvers found the solution, although we note that the (analytical) sensitivity analysis is cheap for many of the problems tested herein. Hence, N_f is presented to indicate the required number of sensitivity analyses, which can be expensive in many structural optimization applications. In such cases, it would be trivial to infer N_f as the measurement of choice.

The results for the test problems outlined in Table 8.1, using the multi-solver approach, are presented in Table 8.2. Tables 8.3, 8.4, 8.5 and 8.6 present results for problem execution using a single solver, which confirms how challenging the test set was, since no single solver successfully solved all problems. From the aforementioned results, it is important to observe that the solver, being *independent* of the Hessian approximation used, has a strong influence on the number of iterations required. `ALGENCAN` failed to solve all the test problems, hence

[3] We solve the Falk dual using the quasi-Newton `l-BFGS-b` solver [4], which efficiently handles the bounds required for the bounded dual [6].

[4] Our 128 GB memory limit was never exceeded in any numerical test.

we do not report on the solver. ALGENCAN merely consumed the available computational resources, although this could not be known *a priori*.

Table 8.2 clearly demonstrates that the effects of NFL on our test set were softened. The solution time required for a multi-start approach, over our selected test problems, is always within a factor of two compared to the fastest single solver implementation. CPLEX, SLA, LSQP and the Falk dual all managed to perform a fastest solution, highlighting the importance of using a diverse set of solvers.

Finally, it is interesting to note the solution time required by the second fastest solver. Often, the second fastest solver required an order of magnitude greater solution time compared to the fastest solver, which may be prohibitively expensive for large-scale problems.

8.6 Conclusions and recommendations

We have proposed a multi-solver, multi-core approach to soften the no-free-lunch (NFL) theorems for optimization, specifically in a structural optimization context. Five different solvers, including the state-of-the-art Galahad LSQP and IBM ILOG CPLEX, simultaneously solved for a wide range of challenging, large-scale structural problems on a multi-core machine. All solvers were required to find the solution for convex and separable quadratic-like subproblems, able to capture both reciprocal and exponential-like behavior, which is desirable in structural optimization.

The test problems contained up to tens of millions of variables and constraints, executed on a single quad-core test platform. The performance impact of running five solvers simultaneously was negligible compared to the cost of using only a single solver, since no single solver successfully solved all the problems tested. Indeed, NFL was softened, with the dual of Falk, LSQP, CPLEX and SLA somewhat alternating in requiring the least solution time.

In future work, we hope to investigate using multiple start points, hopefully ensuring local convergence for many challenging problems. Notwithstanding that we only used three Hessian approximations herein, using a combination of various Hessian approximations could prove fruitful, especially in a multi-physics context, where the physics may not be *a priori* fully understood.

Table 8.1: The large-scale structural test problems.

Number	Problem name	Approximation	n	m	References
1	Vanderplaats' cantilever #1	T2:R	20000000	20000001	[60]
2	Vanderplaats' cantilever #2	T2:R	20000000	20000000	[60]
3	Vanderplaats' cantilever #3	T2:R	20000000	20000000	[60]
4	Semi-infinite #1	SPH-QDR	3	40000	[120]
5	Semi-infinite #3	SPH-QDR	3	40000	[120]
6	Svanberg's snake problem	GRAD	30000	40001	[62]
7	Svanberg's first non-convex problem	SPH-QDR	10000	2	[59]
8	Svanberg's second non-convex problem	SPH-QDR	10000	2	[59]
9	Fleury's weight minimization problem	T2:R	10000000	2	[61]
10	Toropov's cantilever	T2:R	20000000	1	[11, 58]
11	Cam design problem	SPH-QDR	1000	3004	[121]

8.7 Tables for numerical results

Table 8.2: Numerical results for the large-scale structural test problems, using a multi-solver strategy. 'Solver' indicates the first solver to find the problem solution. ‡ indicates that filtered-conservatism [1] was used to enforce global convergence, with the number of inner-iterations, requiring only function evaluations, given in parentheses.

Problem	Solver	f_0^*	h^*	$\mathcal{K}^*$	N_f	Time (s)
1	SLA	6.3640×10^{4}	1.133×10^{-8}	4.861×10^{-6}	21	4.64×10^{2}
2	SLA	5.3714×10^{4}	3.453×10^{-5}	5.076×10^{-8}	22	3.10×10^{2}
3	SLA	5.3714×10^{4}	3.453×10^{-5}	5.076×10^{-8}	22	3.11×10^{2}
4	LSQP	5.3347×10^{0}	0.000×10^{0}	2.213×10^{-5}	9	3.99×10^{0}
5	LSQP	4.3012×10^{0}	7.750×10^{-13}	2.338×10^{-9}	4	3.37×10^{1}
6	SLA	-1.0023×10^{1}	1.268×10^{-5}	8.374×10^{-5}	118	1.48×10^{3}
6‡	CPLEX	-1.0023×10^{1}	0.000×10^{0}	4.383×10^{-5}	35 (22)	1.32×10^{3}
7	SLA	2.6268×10^{3}	1.598×10^{-9}	1.003×10^{-4}	267	7.33×10^{-1}
8	Falk	-7.3732×10^{3}	3.791×10^{-6}	1.059×10^{-4}	2760	4.58×10^{0}
9	SLA	9.5000×10^{10}	4.721×10^{-8}	1.166×10^{0}	21	2.42×10^{1}
10	SLA	1.3197×10^{0}	5.821×10^{-6}	1.892×10^{-5}	12	2.49×10^{1}
11	LSQP	-4.2614×10^{0}	0.000×10^{0}	1.927×10^{-6}	15	2.15×10^{2}

Table 8.3: Standalone solver numerical results, using SLA, for the large-scale structural test problems.

SLA	Problem	f_0^*	h^*	$\mathcal{K}^*$	N_f	Time (s)
	1	6.3640×10^{4}	1.133×10^{-8}	4.861×10^{-6}	21	2.68×10^{2}
	2	5.3714×10^{4}	3.453×10^{-5}	5.076×10^{-8}	22	1.85×10^{2}
	3	5.3714×10^{4}	3.453×10^{-5}	5.076×10^{-8}	22	1.85×10^{2}
	4	†	†	†	†	†
	5	7.3333×10^{4}	0.000×10^{0}	1.915×10^{-2}	33	1.60×10^{2}
	6	-1.0023×10^{1}	1.268×10^{-5}	8.374×10^{-5}	118	1.30×10^{3}
	6‡	-1.0023×10^{1}	5.766×10^{-8}	5.438×10^{-5}	33 (211)	2.77×10^{3}
	7	2.6268×10^{3}	1.598×10^{-9}	1.003×10^{-4}	267	7.15×10^{-1}
	8	†	†	†	†	†
	9	9.5000×10^{10}	4.721×10^{-8}	1.166×10^{0}	21	2.28×10^{1}
	10	1.3197×10^{0}	5.821×10^{-6}	1.892×10^{-5}	12	2.43×10^{1}
	11	—	—	—	—	—

Table 8.4: Standalone solver numerical results, using LSQP, for the large-scale structural test problems.

LSQP	Problem	f_0^*	h^*	$\mathcal{K}^*$	N_f	Time (s)
	1	†	†	†	†	†
	2	5.3714×10^{4}	0.000×10^{0}	7.675×10^{-6}	21	3.14×10^{3}
	3	5.3714×10^{4}	0.000×10^{0}	7.675×10^{-6}	21	3.14×10^{3}
	4	5.3347×10^{0}	0.000×10^{0}	2.213×10^{-5}	9	3.17×10^{0}
	5	4.3012×10^{0}	7.750×10^{-13}	2.338×10^{-9}	4	1.99×10^{1}
	6	-1.0023×10^{1}	5.867×10^{-5}	1.856×10^{-5}	82	1.97×10^{3}
	6‡	-1.0023×10^{1}	9.270×10^{-5}	5.352×10^{-5}	50 (36)	2.02×10^{3}
	7	2.6268×10^{3}	0.000×10^{0}	8.957×10^{-5}	559	1.74×10^{1}
	8	†	†	†	†	†
	9	†	†	†	†	†
	10	1.3197×10^{0}	5.819×10^{-6}	1.891×10^{-5}	12	1.03×10^{3}
	11	-4.2614×10^{0}	0.000×10^{0}	1.927×10^{-6}	15	2.09×10^{2}

Table 8.5: Standalone solver numerical results, using CPLEX, for the large-scale structural test problems.

CPLEX	Problem	f_0^*	h^*	$\mathcal{K}^*$	N_f	Time (s)
	1	6.3640×10^{4}	4.064×10^{-10}	2.692×10^{-5}	19	1.32×10^{4}
	2	5.3714×10^{4}	5.995×10^{-14}	3.746×10^{-7}	24	1.33×10^{4}
	3	5.3714×10^{4}	5.995×10^{-14}	3.746×10^{-7}	24	1.33×10^{4}
	4	†	†	†	†	†
	5	4.3012×10^{0}	0.000×10^{0}	2.033×10^{-5}	4	1.07×10^{4}
	6	†	†	†	†	†
	6‡	-1.0023×10^{1}	0.000×10^{0}	4.383×10^{-5}	35 (22)	1.25×10^{3}
	7	†	†	†	†	†
	8	†	†	†	†	†
	9	†	†	†	†	†
	10	1.3308×10^{0}	0.000×10^{0}	2.661×10^{-5}	12	1.19×10^{3}
	11	—	—	—	—	—

Table 8.6: Standalone solver numerical results, using the Falk dual, for the large-scale structural test problems.

Falk	Problem	f_0^*	h^*	$\mathcal{K}^*$	N_f	Time (s)
	1	†	†	†	†	†
	2	†	†	†	†	†
	3	†	†	†	†	†
	4	5.3347×10^{0}	1.005×10^{-10}	4.641×10^{-6}	8	1.96×10^{1}
	5	4.3012×10^{0}	6.218×10^{-7}	3.170×10^{-6}	5	3.41×10^{2}
	6	†	†	†	†	†
	6‡	†	†	†	†	†
	7	2.6268×10^{3}	2.022×10^{-9}	9.660×10^{-5}	433	1.46×10^{0}
	8	-7.3732×10^{3}	3.791×10^{-6}	1.059×10^{-4}	2760	4.44×10^{0}
	9	—	—	—	—	—
	10	5.4051×10^{0}	0.000×10^{0}	6.977×10^{-5}	10	3.48×10^{1}
	11	†	†	†	†	†

Chapter 9

Conclusion and recommendations

9.1 Conclusions

We have successfully developed two novel primal-dual algorithms, which solve a sequence of convex and separable quadratic-like subproblems. The first solver, `SALA`, is presented in Chapter 4 and contains embarrassingly parallel primal and dual variable expressions, making the algorithm ideal for implementation on massively parallel computational devices, such as GPGPUs. Since `SALA` is able to exploit the intervening variables that are popular in structural optimization, challenging structural design problems were chosen to demonstrate the efficacy of `SALA` against the state-of-the-art Galahad `LSQP` and `ALGENCAN` solvers. Unfortunately, `SALA` contains a troublesome penalty parameter; a result of using an augmented Lagrangian statement for the subproblems. The performance of `SALA` is highly dependent on selecting an optimal penalty parameter value, which in turn is often problem dependent, hence `SALA`'s performance is heavily dependent on problem type. Indeed, the separability inherent to `SALA` is readily exploited on today's GPGPUs, with preliminary GPGPU results indicating a significant reduction in solution time compared to a serial implementation of `SALA`.

The second solver, `SLA`, is developed in Chapter 5. Like `SALA`, `SLA` is able to exploit intervening variables, hence the algorithm is well-suited for structural design problems. `SLA`, based on a Lagrangian statement, contains closed-form expressions for both the primal and dual variables; both of which are readily exploited on massively parallel computational devices. Notwithstanding the parallel scalability of `SLA`, `SLA` is extremely efficient in a serial implementation, hence we provide numerical results for very large-scale structural design problems. Said problems contain up to hundreds of millions of primal variables and constraints, which `SLA` efficiently solves in a few minutes on a single quad-core CPU. Preliminary GPGPU results for `SLA` are encouraging, with more than a fourfold reduction in solution time observed for selected test problems, when compared to a serial implementation.

Since `SLA` is ideally suited to the low-rank SVD problem, we develop two low-rank SVD solvers based on `SLA`. The first solver, `saosvd-d`, is presented in Chapter 6 and contains embarrassingly parallel primal and dual variable updates, hence we present both CPU and GPGPU implementations of the algorithm. For each singular-triplet, `saosvd-d` solves a se-

quence of constrained, convex and quadratic-like subproblems that maximize the Rayleigh quotient. To solve for singular-triplets beyond the first, a Schur complement deflation technique is employed, which removes the variance from previously solved for singular-triplets. Since deflation generally results in a dense matrix, `saosvd-d` is better suited for the low-rank decomposition of dense matrices. Furthermore, no global convergence mechanism is required for `saosvd-d`, as opposed to `saosvd`, which we discuss in what is to follow. Both `saosvd-d` and `saosvd` exploit a novel scaling strategy based on previously solved for singular values, resulting in the construction of few convex subproblems for each singular-triplet. Numerical results demonstrate the efficacy of `saosvd-d` across a wide-range of challenging test problems, compared to state-of-the-art Lanczos methods, in both CPU and GPGPU implementations.

The second low-rank SVD solver, `saosvd`, is presented in Chapter 7. `saosvd` solves a sequence of constrained, convex problems that maximize the Rayleigh quotient, much like `saosvd-d`. However, `saosvd` uses linear orthogonality constraints instead of deflation to solve for singular-triplets beyond the first, hence the algorithm is useful for both dense and sparse matrices. Notwithstanding the use of orthogonality constraints, the variant of `SLA` that `saosvd` exploits results in embarrassingly parallel, closed-form primal and dual variable updates, ideal for GPGPU implementation. Numerical results demonstrate that the CPU implementation of `saosvd` is competitive with state-of-the-art Lanczos methods, while the GPGPU implementation of `saosvd` far outperforms the GPGPU implementation of MATLAB's Lanczos `svds` algorithm.

Finally, in Chapter 8, we propose a strategy to soften the no-free-lunch theorems for structural optimization. For a range of large-scale and challenging structural problems; we demonstrate the efficacy of simultaneously executing multiple solvers, on a multi-core machine, for each respective problem. The mean solution time for the test set was significantly lower with our proposed approach, as opposed to using any of the standalone solver implementations tested. The solver strategies included pure dual, interior-point, active-set and augmented Lagrangian methods, resulting in a heavy dependence of solver performance on problem type, of which there were many in our test set.

9.2 Recommendations for future work

For `SALA`, the penalty parameter scaling issue is most definitely an open ended area of research; as previously mentioned, Lenoir and Mahey dedicated an entire paper to the problem [49]. Although we have briefly experimented with a few of the penalty update strategies in [49], including n scaling parameters for each of the n separable primal problems, many other possibilities exist. Said scaling strategies should additionally be assessed in both a local and global SAOi convergence context, the latter being important in structural optimization. With respect to exploiting the separability of `SALA`, we have clearly demonstrated the efficacy of a CUDA GPGPU implementation. However, an *efficient* GPGPU implementation will need to be assessed to determine the true efficacy of a parallel implementation, in *both* dense and sparse variants. Many of the dense problems tested herein required a prohibitively large number of inner-iterations, since the sparse constraint distribution strategy could not

be exploited. Although we have presented but two constraint distribution strategies herein, further research on alternative dense and sparse strategies is necessary, especially if `SALA` is to be competitive with state-of-the-art methods.

Algorithm `SLA` has two clear areas for future development, both of which can significantly affect the performance of the algorithm. The first area pertains to the active-set strategy used. Despite proposing an efficient and straightforward active-set strategy, many other possibilities exist; hence we recommend different active-set strategies be investigated within the current `SLA` framework. Furthermore, with interior-point methods being somewhat state-of-the-art in solving SQP-like problems[1], it is worth investigating the performance of an interior-point variant of `SLA`. The second area of development is with respect to the primal-projection strategy exploited. The current strategy is inspired by the dual of Falk, hence no dual variables are required for the bound constraints, resulting in minimal computational effort to handle said constraints. However, it is unclear if our primal-projection strategy is optimal, since many strategies exist to handle (simple) bound constraints. Further research into said strategies, that are able to accommodate the salient features of `SLA`, should be conducted to determine the efficacy of the current projection strategy. Additional areas of research, which are agnostic to the aforementioned strategies, include comparisons between different linear solvers and (more) efficient GPGPU implementations for both dense and sparse variants of `SLA`[2].

Both algorithms `saosvd-d` and `saosvd` are sensitive to the initial primal vector chosen. Notwithstanding that our power-method strategy, using uniformly distributed random vectors, is effective; many other possibilities exist to estimate an initial primal vector. Furthermore, instead of the spherical Hessian approximation used herein, both algorithms may benefit from Hessian approximations that are tailor-made for the low-rank SVD problem. Although `saosvd-d` can be implemented in both single and double precision variants, `saosvd` is only currently available in double precision, due to the high precision required for the orthogonality constraints. This warrants an investigation into a mixed-precision variant of `saosvd`, which should reduce both memory requirements and solution time. Additionally for `saosvd`, the performance effect of various global enforcement mechanisms should be investigated, with many alternatives available besides filtered conservatism. Lastly, a brief note on a subspace or block variant of `saosvd`: we have attempted a block variant of `saosvd`, however, there are two main difficulties associated with a block implementation. The first is a loss of embarrassingly parallel dual variable updates, which may or not be prohibitively expensive, depending on the desired decomposition rank. Secondly, choosing a suitable scaling factor becomes difficult, especially when singular values of differing magnitudes exist in the desired subspace. It is critical that the subproblems are well-conditioned through scaling, ensuring the construction of few convex subproblems per singular-triplet. If a suitable scaling strategy is developed for a block `saosvd`, an increased efficiency in performing block BLAS operations should result in reduced solution time, provided the dual linear system is computationally cheap to solve.

[1] Both the state-of-the-art Galahad `LSQP` [51] and IBM ILOG `CPLEX` [116] solvers exploit interior-point methods.

[2] Like `SALA`, this may require handwritten GPGPU kernels instead of BLAS routines, which fully exploit the separability of `SLA`.

Finally, we have presented a single strategy to soften the no-free-lunch theorems for structural optimization. Other strategies include using differing/mixed Hessian approximations for a given problem (especially if the problem physics is not fully understood beforehand) and executing both locally and globally convergent solvers simultaneously. For structural applications, it will be worth assessing the impact of performing multiple sensitivity analyses simultaneously, specifically on multi-core machines with sufficient memory.

List of References

[1] A.A. Groenwold and L.F.P. Etman. On the conditional acceptance of iterates in SAO algorithms based on convex separable approximations. *Struct. Mult. Optim.*, 42:165–178, 2010.

[2] A.A. Groenwold and L.F.P. Etman. Sequential approximate optimization using dual subproblems based on incomplete series expansions. *Struct. Mult. Optim.*, 36:547–570, 2008.

[3] A.A. Groenwold and L.F.P. Etman. *The 'Not-So-Short' manual for the SAOi algorithm.* University of Stellenbosch, Department of Mechanical and Mechatronic Engineering, Stellenbosch, South Africa, 8 December 2011. Version 0.8.7.

[4] C. Zhu, R. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM T. Math. Software (TOMS)*, 23:550–560, 1997.

[5] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.

[6] D.W. Wood, A.A. Groenwold, and L.F.P. Etman. Bounding the dual of Falk to circumvent the requirement of relaxation in globally convergent SAO algorithms. *Optim. Eng.*, 2011.

[7] R. Fletcher, S. Leyffer, and P.L. Toint. On the global convergence of an SLP-filter algorithm. 1998.

[8] K. Svanberg. The method of moving asymptotes - a new method for structural optimization. *Int. J. Numer. Meth. Eng.*, 24:359–373, 1987.

[9] K. Svanberg. A globally convergent version of MMA without linesearch. In G.I.N. Rozvany and N. Olhoff, editors, *Proc. First World Congress on Structural and Multidisciplinary Optimization*, pages 9–16, Goslar, Germany, 1995.

[10] C. Fleury and V. Braibant. Structural optimization: a new dual method using mixed variables. *Int. J. Numer. Meth. Eng.*, 23:409–428, 1986.

[11] A.A. Groenwold and L.F.P. Etman. SAOi: an algorithm for very large scale optimal design. In *Proc. Ninth World Congress on Structural and Multidisciplinary Optimization*, Shizuoka, Japan, June 2011. Paper 035.

[12] J.A. Snyman. *Practical Mathematical Optimization.* Springer, 2005.

[13] R.T Rockafellar. The multiplier method of Hestenes and Powell applied to convex programming. *J. Optim. Theory Appl.*, 12:555–562, 1973.

[14] G.W. Stewart. On the early history of the singular value decomposition. *SIAM Rev.*, 35:551–566, 1993.

[15] C.D. Martin and M.A. Porter. The extraordinary SVD. *Amer. Math. Monthly*, 119:838–851, 2012.

[16] A. Turan and A. Muğan. Structural and sensitivity reanalyses based on singular value decomposition. *Struct. Mult. Optim.*, 48:327–337, 2013.

[17] N. K. Mani, E. J. Haug, and K. E. Atkinson. Application of singular value decomposition for analysis of mechanical system dynamics. *J. Mech. Design*, 107:82–87, 1985.

[18] L De Lathauwer, B De Moor, and J Vandewalle. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl.*, 21:1253–1278, 2000.

[19] Ed F Deprettere, editor. *SVD and Signal Processing: Algorithms, Applications and Architectures*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1988.

[20] R.J. Vaccaro and College of Engineering University of Rhode Island. *SVD and signal processing, II: algorithms, analysis, and applications*. Elsevier, 1991.

[21] M Moonen and B De Moor. *SVD and Signal Processing, III: Algorithms, Architectures and Applications*. Elsevier Science, 1995.

[22] A. Haidar, K. Kabir, D. Fayad, S. Tomov, and J. Dongarra. Out of memory SVD solver for big data. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2017.

[23] W. Yu, W. Li, A. Elsherbeni, and Y. Rahmat-Samii. Singular value decomposition. In *Advanced Computational Electromagnetic Methods and Applications*. Artech House, 2015.

[24] Kenneth J Beers. 3.16.1 SVD Analysis and the Existence/Uniqueness Properties of Linear Systems. In *Numerical Methods for Chemical Engineering - Applications in MATLAB*. Cambridge University Press, 2007.

[25] G.W. Stewart, G.H. Golub, and A. Hoffman. A generalization of the Eckart-Young-Mirsky matrix approximation theorem. *Linear Algebra Appl.*, 88-89:317–327, 1987.

[26] S. Elhay, G.M.L. Gladwell, G.H. Golub, and Y.M. Ram. On some eigenvector-eigenvalue relations. *SIAM J. Matrix Anal. A.*, 20:563–574, 1999.

[27] R.A. Tapia, J.E. Dennis, and J.P. Schäfermeyer. Inverse, shifted inverse, and Rayleigh quotient iteration as Newton's method. *SIAM Rev.*, 60:3–55, 2018.

[28] A. Edelman, T. A Arias, and S.T. Smith. The geometry of algorithms with orthogonality constraints. *SIAM J. Matrix Anal. A.*, 20:303–353, 1998.

[29] N. Halko, P. Martinsson, Y. Shkolnisky, and M. Tygert. An algorithm for the principal component analysis of large data sets. *SIAM. J. Sci. Comput.*, 33:2580–2594, 2011.

[30] J. Baglama and L. Reichel. Augmented implicitly restarted Lanczos bidiagonalization methods. *SIAM. J. Sci. Comput.*, 27:19–42, 2005.

[31] J. Kuczyski and H. Woniakowski. Estimating the largest eigenvalue by the Power and Lanczos algorithms with a random start. *SIAM J. Matrix Anal. A.*, 13:1094–1122, 1992.

[32] A. Edelman and Y. Wang. *Random Matrix Theory and Its Innovative Applications*, pages 91–116. Springer US, Boston, MA, 2013.

[33] B.N. Parlett. The Rayleigh quotient iteration and some generalizations for nonnormal matrices. *Math. Comput.*, 28:679–693, 1974.

[34] P.Y. Papalambros and D.J. Wilde. *Principles of optimal design: modeling and computation*. Cambridge university press, 2000.

[35] K.M. Palanduz and A.A. Groenwold. A separable augmented Lagrangian algorithm for optimal structural design. *Struct. Mult. Optim.*, 61:343–352, 2020.

[36] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, second edition, 2006.

[37] A.R. Conn, N.I. Gould, and P.L. Toint. *LANCELOT: a Fortran package for large-scale nonlinear optimization (Release A)*, volume 17 of *Springer Series in Computational Mathematics*. Springer Verlag, Heidelberg, New York, 1992.

[38] A.A. Groenwold, L.F.P. Etman, and D.W. Wood. Approximated approximations for SAO. *Struct. Mult. Optim.*, 41:39–56, 2010.

[39] M.R. Hestenes. Multiplier and gradient methods. *J. Optim. Theory Appl.*, 4:303–320, 1969.

[40] M.J.D. Powell. A method for nonlinear constraints in optimization. In R. Fletcher, editor, *Optimization*, pages 283–298. Academic Press, New York, 1969.

[41] A. Hamdi, P. Mahey, and J.P. Dussault. A new decomposition method in nonconvex programs via separable augmented Lagrangians. In P. Gritzman, R. Horst, E. Sachs, and R. Tichatschke, editors, *Recent Advances in Optimization*, volume 452 of *Lecture notes in Economics and Mathematical Systems*. Springer, 1997.

[42] A. Hamdi and P. Mahey. Separable diagonalized multiplier method for decomposing nonlinear programs. *Comput. Appl. Math.*, 19:1–29, 2000.

[43] A. Hamdi. Two-level primal-dual proximal decomposition technique to solve large scale optimization problems. *Appl. Math. and Comput.*, 160:921–938, 2005.

[44] A. Hamdi. A primal-dual proximal point algorithm for constrained convex programs. *Appl. Math. and Comput.*, 162:293–303, 2005.

[45] A. Hamdi. Decomposition for structured convex programs with smooth multiplier methods. *Appl. Math. and Comput.*, 169:218–241, 2005.

[46] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 3:1–122, 2010.

[47] J. Douglas and H.H. Rachford. On the numerical solution of heat conduction problems in two and three space variables. *Trans. Amer. Math. Soc.*, 82:421–439, 1956.

[48] P.L. Lions and B. Mercier. Splitting algorithms for the sum of two nonlinear operators. *SIAM J. Control Optim.*, 22:277–293, 1984.

[49] A. Lenoir and P. Mahey. Global and adaptive scaling in a separable augmented Lagrangian algorithm. Research Report LIMOS RR-07-14, Université Blaise Pascal, Clermont-Ferrand, 2007.

[50] E.G. Birgin and J.M. Martínez. *Practical Augmented Lagrangian Methods for Constrained Optimization*. Fundamentals of Algorithms. SIAM, Philadelphia, Pennsylvania, 2007.

[51] N.I.M. Gould, D. Orban, and Ph. L. Toint. GALAHAD, a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization. *CM Transactions on Mathematical Software*, 29:353–372, 2003.

[52] L.F.P. Etman, A.A. Groenwold, and J.E. Rooda. On diagonal QP subproblems for sequential approximate optimization. In *Proc. Eighth World Congress on Structural and Multidisciplinary Optimization*, Lisboa, Portugal, June 2009. Paper 1065.

[53] L.F.P. Etman, A.A. Groenwold, and J.E. Rooda. First-order sequential convex programming using approximate diagonal QP subproblems. *Struct. Mult. Optim.*, 45:479–488, 2012.

[54] J.A. Snyman and A.M. Hay. The Dynamic-Q optimization method: an alternative to SQP? *Comput. Math. Appl.*, 44:1589–1598, 2002.

[55] D.N. Wilke, S. Kok, and A.A. Groenwold. The application of gradient-only optimization methods for problems discretized using non-constant methods. *Struct. Mult. Optim.*, 40:433–451, 2010.

[56] A.A. Groenwold and L.F.P. Etman. A quadratic approximation for structural topology optimization. *Int. J. Numer. Meth. Eng.*, 82:505–524, 2010.

[57] Y. Kanno and S. Kitayama. Alternating direction method of multipliers as a simple effective heuristic for mixed-integer nonlinear optimization. *Struct. Mult. Optim.*, 58:1291–1295, 2018.

[58] V.V. Toropov. Personal discussion, November 2008.

[59] K. Svanberg. A class of globally convergent optimization methods based on conservative convex separable approximations. *SIAM J. Optim.*, 12:555–573, 2002.

[60] G.N. Vanderplaats. *Numerical optimization techniques for engineering design*. Vanderplaats R&D, Inc., Collorado Springs, 2001.

[61] C. Fleury. Structural weight optimization by dual methods of convex programming. *Int. J. Numer. Meth. Eng.*, 14:1761–1783, 1979.

[62] K. Svanberg. On a globally convergent version of MMA. In *Proc. Seventh World Congress on Structural and Multidisciplinary Optimization*, Seoul, Korea, May 2007. Paper no. A0052.

[63] E.D. Dolan, J.J. Moré, and T.S. Munson. Benchmarking optimization software with COPS 3.0. Mathematics and Computer Science Division Technical Report ANL/MCS-TM-273, Argonne National Laboratory, Illinois, U.S.A., February 2004.

[64] W. Hock and K. Schittkowski. *Test examples for nonlinear programming codes*, volume 187 of *Lecture notes in Economics and Mathematical Systems*. Springer-Verlag, Berlin, Heidelberg, New York, 1981.

[65] D. Wolpert and W. Macready. No free lunch theorems for optimization. *IEEE Trans. Evol. Comput.*, 1:67–82, 1997.

[66] P. Duysinx, W.H. Zhang, C. Fleury, V.H. Nguyen, and S. Haubruge. A new separable approximation scheme for topological problems and optimization problems characterized by a large number of design variables. In N. Ollhoff and G.I.N. Rozvany, editors, *Proc. First World Congress on Structural and Multidisciplinary Optimization*, pages 1–8, Goslar, Germany, 1995.

[67] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5:46–55, 1998.

[68] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6:40–53, 2008.

[69] K.M. Palanduz and A.A. Groenwold. A separable primal-dual algorithm for very large scale optimal structural design. *Struct. Mult. Optim.*, 2020. (Under Review).

[70] J.E. Falk. Lagrange multipliers and nonlinear programming. *J. Math. Anal. Appls.*, 19:141–159, 1967.

[71] A.A. Groenwold. On the linearization of separable quadratic constraints in dual sequential convex programs. *Comput. Struct.*, 102:42–48, 2012.

[72] G. Hadley. *Nonlinear and Dynamic Programming*. Addison-Wesley, Reading, Massachusetts, 1964.

[73] J.E. Falk. *A constrained Lagrangian approach to nonlinear programming*. PhD thesis, University of Michigan, 1965.

[74] D.W. Wood and A.A. Groenwold. On convex transformability and the solution of nonconvex problems via the dual of Falk. *Struct. Mult. Optim.*, 46:171–185, 2012.

[75] R.T. Haftka and Z. Gürdal. *Elements of structural optimization*, volume 11. Springer Science & Business Media, 2012.

[76] C. Fleury. Structural optimization methods for large scale problems: computational time issues. In *Proceedings eighth world congress on structural and multidisciplinary optimization, Lisbon*, volume 1184, June 2009.

[77] M. Hintermüller, K. Ito, and K. Kunisch. The primal-dual active set strategy as a semismooth Newton method. *SIAM J. Optim.*, 13:865 – 888, 2002.

[78] A.A. Groenwold, D.W. Wood, L.F.P. Etman, and S. Tosserams. Globally convergent optimization algorithm using conservative convex separable diagonal quadratic approximations. *AIAA J.*, 47:2649–2657, 2009.

[79] R. Fletcher and S. Leyffer. User manual for filterSQP. Numerical Analysis Report NA\181, Department of Mathematics, University of Dundee, Dundee, Schotland, April 1998.

[80] R. Fletcher and S. Leyffer. Nonlinear programming without a penalty function. *Math. Program.*, 91:239–269, 2002.

[81] R. Fletcher, S. Leyffer, and P.L. Toint. On the global convergence of an SLP-filter algorithm. Technical Report 00/15, Department of Mathematics, University of Namur, Namur, Belgium, 1998.

[82] R. Fletcher, S. Leyffer, and P.L. Toint. On the global convergence of a filter–SQP algorithm. *SIAM J. Optim.*, 13:44–59, 2002.

[83] M.J.D. Powell. The Lagrange method and SAO with bounds on the dual variables. *Optim. Method Softw.*, 29:224–238, 2014.

[84] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[85] L.S. Blackford, A. Petitet, R. Pozo, K. Remington, R.C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (BLAS). *ACM T. Math. Software*, 28:135–151, 2002.

[86] A. De Coninck, B. De Baets, D. Kourounis, F. Verbosio, O. Schenk, S. Maenhout, and J. Fostier. Needles: Toward large-scale genomic prediction with marker-by-environment interaction. 203:543–555, 2016.

[87] F. Verbosio, A. De Coninck, D. Kourounis, and O. Schenk. Enhancing the scalability of selected inversion factorization algorithms in genomic prediction. *J. Comput. Sci.*, 22:99 – 108, 2017.

[88] D. Kourounis, A. Fuchs, and O. Schenk. Towards the next generation of multiperiod optimal power flow solvers. *IEEE Trans. Power Syst.*, PP:1–10, 2018.

[89] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations - Version 2, 1994.

[90] O. Sigmund. A 99 line topology optimization code written in Matlab. *Struct. Mult. Optim.*, 21:120–127, 2001.

[91] L. Krog, M. Bruyneel, A. Remouchamps, and C. Fleury. COMBOX: a distributed computing process for optimum pre-sizing of composite aircraft box structures. volume 1314, 2007.

[92] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24:235–265, 1997.

[93] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. Fast Cholesky factorization on GPUs for batch and native modes in MAGMA. *J. Comput. Sci.*, 20:85 – 93, 2017.

[94] K.M. Palanduz and A.A. Groenwold. A GPGPU accelerated, constrained and separable convex optimization approach for low-rank singular value decomposition. *J. Comput. Sci.*, 2020. (Under Review).

[95] D. Calvetti, L. Reichel, and D.C. Sorensen. An implicitly restarted Lanczos method for large symmetric eigenvalue problems. *Electron. T. Numer. Ana.*, 2:21, 1994.

[96] C. Yang. Solving large-scale eigenvalue problems in SciDAC applications. In *Journal of Physics: Conference Series*, volume 16, page 425. IOP Publishing, 2005.

[97] C. Vömel, S.Z. Tomov, O.A. Marques, A. Canning, L. Wang, and J.J. Dongarra. State-of-the-art eigensolvers for electronic structure calculations of large scale nano-systems. *J. Comput. Phys.*, 227:7113–7124, 2008.

[98] L. Wu, F. Xue, and A. Stathopoulos. TRPL+K: Thick-restart preconditioned Lanczos+K method for large symmetric eigenvalue problems. *SIAM. J. Sci. Comput.*, 41:1013–1040, 2019.

[99] M.W. Berry. *Multiprocessor sparse SVD algorithms and applications*. PhD thesis, University of Illinois, 1991.

[100] K.K. Matam and K. Kothapalli. GPU accelerated Lanczos algorithm with applications. In *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, pages 71–76. IEEE, 2011.

[101] J.M. Cavanagh, T.E. Potok, and X. Cui. Parallel latent semantic analysis using a graphics processing unit. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2505–2510, 2009.

[102] D. Garber and E. Hazan. Fast and simple PCA via convex optimization. *arXiv preprint arXiv:1509.05647*, 2015.

[103] J. Wright, A. Ganesh, S. Rao, Y. Peng, and Y. Ma. Robust principal component analysis: Exact recovery of corrupted low-rank matrices via convex optimization. In *Advances in neural information processing systems*, pages 2080–2088, 2009.

[104] L.W. Mackey. Deflation methods for sparse PCA. In *Advances in neural information processing systems*, pages 1017–1024, 2009.

[105] N. Muller, L. Magaia, and B.M. Herbst. Singular value decomposition, eigenfaces, and 3D reconstructions. *SIAM Rev.*, 46:518–545, 2004.

[106] R. Larsen. PROPACK-software for large and sparse SVD calculations, version 2.1, 2005.

[107] T.A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38:1–25, 2011.

[108] I.S. Duff, R.G. Grimes, and J.G. Lewis. *Users' guide for the Harwell-Boeing sparse matrix collection (Release I)*, 1992.

[109] K.M. Palanduz and A.A. Groenwold. A sequential approximate optimization approach for low-rank singular value decomposition. *Adv. Eng. Softw.*, 2020. (Submitted).

[110] M.W. Berry. Large-scale sparse singular value computations. *Int. J. Supercomput. Appl.*, 6:13–49, 1992.

[111] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *J. Assoc. Inf. Sci. Techno.*, 41:391–407, 1990.

[112] F. Götze and A. Tikhomirov. Rate of convergence in probability to the Marchenko-Pastur law. *Bernoulli*, 10:503–548, 06 2004.

[113] K.M. Palanduz and A.A. Groenwold. Softening the no-free-lunch theorems for structural optimization. *Int. J. Numer. Meth. Eng.*, 2020. (Submitted).

[114] R.T. Haftka and Z. Gürdal. *Elements of structural optimization*, volume 11 of *Solid Mechanics and its applications*. Kluwer Academic Publishers, Dordrecht, the Netherlands, third edition, 1991.

[115] A.A. Groenwold, L.F.P. Etman, J.A. Snyman, and J.E. Rooda. Incomplete series expansion for function approximation. *Struct. Mult. Optim.*, 34:21–40, 2007.

[116] IBM ILOG CPLEX. *V12.7.1 User's Manual for CPLEX 2018*, 2018.

[117] TANGO. *ALGENCAN 3.1.1*, 2017.

[118] A.A. Groenwold, L.F.P. Etman, S. Kok, D.W. Wood, and S. Tosserams. An augmented Lagrangian approach to non-convex SAO using diagonal quadratic approximations. *Struct. Mult. Optim.*, 38:415–421, 2009.

[119] O. Tange. GNU Parallel, July 2020.

[120] Tanaka Y., Fukushima M., and Ibaraki T. A comparative study of several semi-infinite nonlinear programming algorithms. *European J. Oper. Research*, 36:92–100, 1988.

[121] E.D. Dolan, J.J. Moré, and T.S. Munson. Benchmarking optimization software with cops 3.0. Technical report, Argonne National Lab., Argonne, IL (US), 2004.